ȘTEFAN TANASĂ, ȘTEFAN ANDREI, CRISTIAN OLARU

Java

de la 0 la expert

- CLASE, INTERFEȚE ȘI TABLOURI
- FIȘIERE ȘI FLUXURI DE DATE
- APPLETURI
- BAZE DE DATE
- SERVLETURI

POLIROM

ŞTEFAN TANASĂ, ŞTEFAN ANDREI,
CRISTIAN OLARU

Java de la 0 la expert

Seria *Web* este coordonată de dr. Sabin Buraga
(Facultatea de Informatică, Universitatea „Al.I. Cuza", Iaşi)

© 2003, 2007, 2011 by Editura POLIROM

Această carte este protejată prin copyright. Reproducerea integrală sau parţială, multiplicarea prin orice mijloace şi sub orice formă, cum ar fi xeroxarea, scanarea, transpunerea în format electronic sau audio, punerea la dispoziţia publică, inclusiv prin internet sau prin reţele de calculatoare, stocarea permanentă sau temporară pe dispozitive sau sisteme cu posibilitatea recuperării informaţiilor, cu scop comercial sau gratuit, precum şi alte fapte similare săvârşite fără permisiunea scrisă a deţinătorului copyrightului reprezintă o încălcare a legislaţiei cu privire la protecţia proprietăţii intelectuale şi se pedepsesc penal şi/sau civil în conformitate cu legile în vigoare.

www.polirom.ro

Editura POLIROM
Iaşi, B-dul Carol I nr. 4; P.O. BOX 266, 700506
Bucureşti, Splaiul Unirii nr. 6, bl. B3A, sc. 1, et. 1, sector 4,
040031, O.P. 53, C.P. 15-728

Descrierea CIP a Bibliotecii Naţionale a României:

TANASĂ, ŞTEFAN

Java : de la 0 la expert / Ştefan Tanasă, Ştefan Andrei, Cristian Olaru. – Ed. a II-a rev. – Iaşi : Polirom, 2011

ISBN: 978-973-46-2405-8

I. Andrei, Ştefan
II. Olaru, Cristian

004.43 JAVA

Printed in ROMANIA

ŞTEFAN TANASĂ, ŞTEFAN ANDREI, CRISTIAN OLARU

Java

de la 0 la expert

Ediţia a II-a
revăzută şi adăugită

POLIROM
2011

Ştefan Tanasă este cadru didactic la Facultatea de Informatică a Universităţii „Al.I. Cuza" din Iaşi. A predat cursuri de *Proiectarea siturilor Web* şi predă în prezent cursul de *Reţele de calculatoare*. De-a lungul timpului a ţinut ore de laborator la disciplinele *Tehnologii Web*, *Sisteme de operare* şi *Reţele de calculatoare*. Domeniile principale de interes sunt mediile virtuale distribuite, tehnologiile Web şi programarea în Java. La Editura Polirom a mai publicat: *Programare Web în bash şi Perl* (în colab., 2002) şi *Dezvoltarea aplicaţiilor Web folosind JAVA* (în colab., 2005).

Ştefan Andrei este cadru didactic la Facultatea de Informatică a Universităţii „Al.I. Cuza" din Iaşi. A predat cursuri de programare (limbajele C, C++, Java şi Prolog), *Construcţia compilatoarelor*, *Limbaje formale*, *Navigare Internet* şi *Sisteme distribuite*. A obţinut doctoratul în domeniul informaticii cu teza *Procesarea paralelă*, susţinută la Universitatea din Hamburg (Germania). Este preocupat de verificarea şi depanarea sistemelor în timp real şi procesarea datelor în diverse limbaje de programare. A obţinut premiile DAAD (1997), World Bank (1998-2000) şi Singapore – MIT Alliance (2002-2005).

Cristian Olaru a ocupat în decursul timpului diverse poziţii de dezvoltator Java în companii naţionale şi internaţionale. A absolvit Facultatea de Informatică din cadrul Universităţii „Al.I Cuza" din Iaşi şi un masterat în *Sisteme distribuite* la aceeaşi facultate. Consultant al firmelor interesate de dezvoltarea de produse software, folosind Java drept limbaj de programare, este interesat de arhitecturi J2EE, *design patterns*, ORM *frameworks*, Web *frameworks*, servicii Web. La Editura Polirom a mai publicat: *Dezvoltarea aplicaţiilor Web folosind JAVA* (în colab., 2005).

Cuprins

Cuvânt înainte (Dan Grigoraş) 15

Prefaţă la ediţia a II-a 17

1. INTRODUCERE 19
    - 1.1. **Cuvinte-cheie** 19
    - 1.2. **Istoricul şi caracteristicile limbajului Java** 20
    - 1.3. **Instalare** 21
    - 1.4. **Tipuri de aplicaţii Java** 23
    - 1.5. **Maşina virtuală Java** 29
    - 1.6. **Concluzii** 31
    - 1.7. **Test-grilă** 32
    - 1.8. **Exerciţii propuse spre implementare** 32

2. FUNDAMENTELE LIMBAJULUI JAVA 33
    - 2.1. **Cuvinte-cheie** 33
    - 2.2. **Fişiere-sursă** 34
    - 2.3. **Atomi lexicali** 35
        - 2.3.1. Caractere Unicode 36
        - 2.3.2. Traduceri lexicale 37
    - 2.4. **Tipuri de date** 39
    - 2.5. **Expresii şi operatori** 46
    - 2.6. **Variabile** 64
    - 2.7. **Declaraţii şi iniţializări** 69
    - 2.8. **Conversii** 72
        - 2.8.1. Contextele de conversie 76
    - 2.9. **Structuri de control** 82
        - 2.9.1. Instrucţiunea vidă 83
        - 2.9.2. Instrucţiunea etichetă 83
        - 2.9.3. Instrucţiunea expresie 84
        - 2.9.4. Instrucţiunea `if` 84
        - 2.9.5. Instrucţiunea `switch` 85
        - 2.9.6. Instrucţiunea `while` 88
        - 2.9.7. Instrucţiunea `do` 90
        - 2.9.8. Instrucţiunea `for` 92
        - 2.9.9. Instrucţiunea `break` 96

2.9.10. Instrucţiunea `continue` ... 97
2.9.11. Instrucţiunea `return` ... 98
2.9.12. Instrucţiunea `throw` ... 99
2.9.13. Instrucţiunea `synchronized` ... 101
2.9.14. Instrucţiunea `try-catch-finally` ... 102
2.10. Concluzii ... 107
2.11. Test-grilă ... 109
2.12. Exerciţii propuse spre implementare ... 115
2.13. Proiecte propuse spre implementare ... 116

3. CLASE, INTERFEŢE ŞI TABLOURI ... 119

3.1. Cuvinte-cheie ... 119
3.2. Clase ... 120
3.2.1. Domeniul de vizibilitate al numelui unei clase ... 121
3.2.2. Modificatorii unei clase ... 123
3.2.3. Clase derivate ... 126
3.2.4. Clasa `Object` ... 127
3.2.5. Implementarea interfeţelor ... 130
3.2.6. Declaraţiile membrilor unei clase ... 133
3.2.6.1. *Niveluri de acces* ... 135
3.2.6.2. *Declaraţiile atributelor unei clase* ... 140
3.2.6.3. *Declaraţiile metodelor unei clase* ... 151
3.2.7. Iniţializatori statici ... 163
3.2.8. Declaraţiile constructorilor ... 163
3.2.9. Clase interioare ... 169
3.2.10. Clase generice ... 173
3.2.11. Enumerări ... 176
3.2.12. Erori şi excepţii ... 180
3.2.13. Distrugerea obiectelor şi eliberarea memoriei ... 183
3.2.14. Divizarea unei aplicaţii în fişiere ... 187
3.3. Interfeţe ... 190
3.3.1. Declaraţiile atributelor unei interfeţe ... 191
3.3.2. Declaraţiile metodelor unei interfeţe ... 194
3.3.3. Moştenire multiplă prin intermediul interfeţelor ... 196
3.4. Tablouri ... 201
3.5. Conversii ale tipului referinţă ... 209
3.5.1. Conversii implicite ale tipului referinţă ... 209
3.5.2. Conversii explicite ale tipului referinţă ... 211
3.5.3. Conversiile implicite de la tipurile primitive la clasele *wrapper* ... 217
3.5.4. Conversiile implicite de la clasele *wrapper* la tipurile primitive ... 218
3.6. Concluzii ... 218
3.7. Test-grilă ... 219
3.8. Exerciţii propuse spre implementare ... 239

4. ŞIRURI ŞI STRUCTURI DINAMICE DE DATE ... 243

4.1. Cuvinte-cheie ... 243
4.2. Şiruri de caractere ... 244

4.2.1. Interfața `CharSequence` 244
4.2.2. Clasa `String` 245
4.2.3. Clasa `StringBuffer` 251
4.2.4. Clasa `StringTokenizer` 255
4.3. **Structuri dinamice de date** 257
4.3.1. Interfața `Enumeration` 257
4.3.2. Iteratori 258
4.3.3. Colecții 259
4.3.4. Tabele de asocieri 260
4.3.5. Compararea obiectelor 264
4.3.5.1. *Interfața* `Comparable` 264
4.3.5.2. *Interfața* `Comparator` 265
4.3.6. Mulțimi 267
4.3.7. Liste 268
4.3.8. Clasa `BitSet` 273
4.3.9. Clasa `Properties` 275
4.4. **Concluzii** 280
4.5. **Test-grilă** 280
4.6. **Exerciții propuse spre implementare** 281

5. FIȘIERE, FLUXURI DE DATE ȘI SERIALIZAREA OBIECTELOR 283

5.1. **Cuvinte-cheie** 283
5.2. **Introducere** 284
5.3. **Clase Java pentru intrări și ieșiri** 286
5.3.1. Clasa `File` 287
5.3.2. Clasa `RandomAccessFile` 289
5.3.3. Clase Java pentru fluxuri de intrare/ieșire 292
5.3.3.1. *Fluxuri de nivel scăzut* 293
5.3.3.2. *Fluxuri filtru de nivel înalt* 296
5.3.3.3. *Fluxuri pentru citire și fluxuri pentru scriere* 299
5.3.3.4. *Clasele* `PrintWriter` *și* `BufferedReader` 301
5.3.3.5. *Clasa* `StreamTokenizer` 307
5.3.3.6. *Clasa* `System` 313
5.4. **Serializarea obiectelor** 317
5.4.1. Clasa `ObjectOutputStream` 317
5.4.2. Clasa `ObjectInputStream` 320
5.4.3. Interfața `Serializable` 322
5.4.4. Interfața `Externalizable` 323
5.4.5. Crearea unei clase serializabile 324
5.4.5.1. *Derivarea unei clase serializabile* 324
5.4.5.2. *Implementarea interfeței* `Serializable` 325
5.4.5.3. *Metode de serializare obișnuite* 326
5.4.5.4. *Implementarea interfeței* `Externalizable` 330
5.4.5.5. *Declararea interfeței și a tuturor metodelor de acces* 331
5.5. **Concluzii** 336
5.6. **Test-grilă** 336
5.7. **Exerciții propuse spre implementare** 340
5.8. **Proiecte propuse spre implementare** 341

- 6. FIRE DE EXECUȚIE 343
  - 6.1. **Cuvinte-cheie** 343
  - 6.2. **Introducere** 344
    - 6.2.1. Breviar teoretic 344
    - 6.2.2. Corectitudinea programelor 344
    - 6.2.3. Primitivele de programare concurentă 345
  - 6.3. **Programare concurentă în Java** 346
    - 6.3.1. Crearea unui fir de execuție prin extinderea clasei `Thread` 347
    - 6.3.2. Crearea unui *thread* utilizând interfața `Runnable` 348
    - 6.3.3. Controlul unui fir de execuție 350
    - 6.3.4. Prioritatea firelor de execuție 355
    - 6.3.5. Grupurile de fire de execuție 359
    - 6.3.6. Excluderea mutuală și sincronizarea 361
    - 6.3.7. Metodele `wait()` și `notify()` 363
  - 6.4. **Studii de caz: problema filosofilor și problema producător-consumator** 372
    - 6.4.1. Problema filosofilor 372
    - 6.4.2. Problema producător-consumator 377
  - 6.5. **Concluzii** 382
  - 6.6. **Test-grilă** 382
  - 6.7. **Exerciții propuse spre implementare** 383
  - 6.8. **Proiecte propuse spre implementare** 384
- 7. APPLETURI ȘI INSTRUMENTE DE LUCRU CU FERESTRELE (AWT) 385
  - 7.1. **Cuvinte-cheie** 385
  - 7.2. **Appleturi** 386
    - 7.2.1. Clasa `Applet (java.applet.Applet)` 386
    - 7.2.2. Obținerea parametrilor de intrare ai appleturilor 389
    - 7.2.3. Contextul unui applet 391
    - 7.2.4. Comunicarea dintre appleturi 392
    - 7.2.5. Redarea sunetelor și imaginilor 394
      - 7.2.5.1. *Afișarea conținutului grafic* 394
      - 7.2.5.2. *Redarea sunetelor* 396
    - 7.2.6. Transformarea unui applet într-o aplicație de sine stătătoare 398
  - 7.3. **Grafica în Java** 399
  - 7.4. **Componente și evenimente** 404
    - 7.4.1. Bucla evenimentelor 404
    - 7.4.2. Componente 408
    - 7.4.3. Containere 410
    - 7.4.4. Etichete 410
    - 7.4.5. Butoane 412
    - 7.4.6. Câmpuri și arii text 414
      - 7.4.6.1. *Clasa* `java.awt.TextComponent` 414
      - 7.4.6.2. *Clasa* `java.awt.TextField` 415
      - 7.4.6.3. *Clasa* `java.awt.TextArea` 416
    - 7.4.7. Butoane de selecție (*checkbox* și radio) 421
    - 7.4.8. Liste de opțiuni 423

7.4.9. Suprafețe de desenare 428
7.4.10. Panouri (*panels*) 429
7.4.11. Meniuri *pop-up* 429

7.5. Gestionari de poziționare (*Layout Managers*) 432
7.5.1. Gestionarul `BorderLayout` 433
7.5.2. Gestionarul `CardLayout` 436
7.5.3. Gestionarul `FlowLayout` 438
7.5.4. Gestionarul `GridLayout` 439
7.5.5. Gestionarul `GridBagLayout` 440
7.5.6. Poziționarea absolută 442

7.6. Ferestre 444
7.6.1. Clasa `java.awt.Window` 444
7.6.2. Clasa `java.awt.Frame` 445
7.6.3. Clasa `java.awt.Dialog` 447
7.6.4. Clasa `java.awt.FileDialog` 449

7.7. Managerul de securitate 451

7.8. Concluzii 453

7.9. Test-grilă 453

7.10. Exerciții propuse spre implementare 455

7.11. Proiecte propuse spre implementare 457

8. ACCESUL LA BAZE DE DATE FOLOSIND JDBC 459

8.1. Cuvinte-cheie 459

8.2. Introducere 460

8.3. Clasificarea driverelor JDBC 462

8.4. Breviar SQL 463
8.4.1. Instrucțiunea `CREATE TABLE` 464
8.4.2. Instrucțiunea `INSERT` 464
8.4.3. Instrucțiunea `UPDATE` 465
8.4.4. Instrucțiunea `DELETE` 465
8.4.5. Instrucțiunea `SELECT` 466

8.5. Asocieri de tipuri între SQL și Java 466

8.6. Accesarea unei baze de date folosind JDBC 469
8.6.1. Înregistrarea driverului JDBC folosind clasa `DriverManager` 469
8.6.2. Stabilirea unei conexiuni cu baza de date 470
8.6.3. Execuția unei instrucțiuni SQL 471
8.6.4. Procesarea rezultatelor 474
8.6.5. Închiderea unei conexiuni la o bază de date 478

8.7. Instalarea și configurarea MySQL 479
8.7.1. Introducere 479
8.7.2. Drivere JDBC pentru MySQL 480

8.8. Utilizarea punții JDBC-ODBC 483

8.9. Tranzacții 486
8.9.1. Motivații pentru folosirea tranzacțiilor 486
8.9.2. ACID 488
8.9.3. Niveluri de izolare 489

8.10. Concluzii 490

8.11. Test-grilă 490
8.12. Exerciţii propuse spre implementare 492
8.13. Proiecte propuse spre implementare 493

9. PROGRAMAREA REŢELELOR 495

9.1. Cuvinte-cheie 495
9.2. Introducere 496
  9.2.1. Adrese, porturi şi socketuri 496
9.3. Programarea reţelelor prin intermediul conexiunilor 499
  9.3.1. Clasa `ServerSocket` 499
  9.3.2. Clasa `Socket` 500
  9.3.3. O aplicaţie simplă client/server orientată conexiune 502
9.4. Programarea reţelelor prin intermediul datagramelor 507
  9.4.1. Clasa `DatagramPacket` 507
  9.4.2. Clasa `DatagramSocket` 509
  9.4.3. O aplicaţie simplă client/server neorientată conexiune 511
  9.4.4. Clasa `InetAddress` 514
  9.4.5. Clasa `URL` 516
  9.4.6. Obţinerea unei pagini Web prin intermediul unui socket 522
9.5. Apelul metodelor la distanţă 526
  9.5.1. Obiectul de la distanţă 527
  9.5.2. Registrul de nume 529
  9.5.3. Exemplu de utilizare a apelurilor la distanţă 529
  9.5.4. Localizarea fişierelor RMI. Pachete necesare 532
  9.5.5. Clasa `Naming` 533
  9.5.6. Clasa `LocateRegistry` 534
  9.5.7. Interfaţa `Registry` 534
  9.5.8. Clasa `RemoteObject` 535
  9.5.9. Obiecte la distanţă ca parametri 535
  9.5.10. Serializarea unui obiect la distanţă 537
  9.5.11. Clasa `RemoteServer` 537
  9.5.12. Clasa `RemoteStub` 538
  9.5.13. Interfaţa `Unreferenced` 538
  9.5.14. Clasa `RMISocketFactory` 538
  9.5.15. Interfaţa `RMIFailureHandler` 540
  9.5.16. Exemplu de aplicaţie `RMI` 540
9.6. Concluzii 545
9.7. Test-grilă 545
9.8. Exerciţii propuse spre implementare 547
9.9. Proiecte propuse spre implementare 547

10. SERVLETURI 549

10.1. Cuvinte-cheie 549
10.2. CGI (*Common Gateway Interface*) 550
  10.2.1. Variabilele de mediu 551
  10.2.2. Apelarea programelor CGI din formularele HTML 553

- 10.2.3. Procesarea datelor din formularele HTML prin metoda GET 555
- 10.2.4. Procesarea unui formular prin metoda POST 556

10.3. **Servleturi** 558
- 10.3.1. Servlet API 559
- 10.3.2. Funcționalitatea servleturilor 559
- 10.3.3. Execuția servleturilor 560
- 10.3.4. Structura de bază a unui servlet 561
- 10.3.5. Ciclul de viață al unui servlet 562
- 10.3.6. Clasa `HttpServlet` 564
- 10.3.7. Redirectarea cererii 571
- 10.3.8. Returnarea unei erori 571
- 10.3.9. Exemplu de aplicație Web 572

10.4. Concluzii 583

10.5. Test-grilă 584

10.6. **Exerciții propuse spre implementare** 585

10.7. **Proiecte propuse spre implementare** 585

11. JAVA SERVER PAGES (JSP) 587

11.1. Cuvinte-cheie 587

11.2. Introducere 588

11.3. Elemente JSP 589
- 11.3.1. Comentarii 590
- 11.3.2. Directive 591
  - 11.3.2.1. *Directiva* `page` 592
  - 11.3.2.2. *Directiva* `include` 593
  - 11.3.2.3. *Directiva* `taglib` 594
- 11.3.3. Declarații 594
- 11.3.4. Inițializarea și terminarea unui JSP 595
- 11.3.5. Obiecte implicite 596
- 11.3.6. Expresii 597
- 11.3.7. Scriptlet-uri 597
- 11.3.8. Acțiuni 598
  - 11.3.8.1. *Integrarea componentelor JavaBean* 598

11.4. Concluzii 600

11.5. Test-grilă 600

11.6. **Exerciții propuse spre implementare** 601

11.7. **Proiecte propuse spre implementare** 601

12. **PROCESAREA DOCUMENTELOR XML** 603

12.1. **Cuvinte-cheie** 603

12.2. **Standarde și specificații** 604

12.3. **Instalare** 605

12.4. **SAX** 605
- 12.4.1. Pachete necesare 605
- 12.4.2. Exemplu de document XML 605
- 12.4.3. Obținerea informațiilor din documentele XML 606

12.4.4. Obţinerea valorilor atributelor unui tag ... 610
12.4.5. Procesarea documentelor XML care conţin spaţii de nume ... 611
12.4.6. Procesarea documentelor XML care trebuie validate prin DTD ... 614
12.4.7. Procesarea documentelor XML care trebuie validate via XML Schema ... 617

12.5. DOM ... 619
12.5.1. Pachete necesare ... 620
12.5.2. Crearea arborelui asociat unui document XML ... 621
12.5.3. Manipularea arborelui DOM ... 621
12.5.3.1. *Interfaţa* `Node` ... 621
12.5.3.2. *Procesarea documentelor XML care utilizează spaţii de nume* ... 628
12.5.3.3. *Procesarea documentelor XML care conţin DTD* ... 631
12.5.3.4. *Validarea documentelor XML prin XML Schema* ... 634
12.5.3.5. *Interfaţa* `Document` ... 635

12.6. XSLT ... 638
12.6.1. Pachete necesare ... 638
12.6.2. Realizarea unor transformări ... 639

12.7. Concluzii ... 643

12.8. Test-grilă ... 644

12.9. Exerciţii propuse spre implementare ... 644

12.10. Proiecte propuse spre implementare ... 645

13. INTERACŢIUNEA CU UTILIZATORUL PRIN SWING ... 647

13.1. Cuvinte-cheie ... 647

13.2. Introducere ... 648
13.2.1. Obiectivele urmărite în acest capitol ... 648
13.2.2. Despre interfeţe grafice ... 649
13.2.3. JFC (*Java Foundation Classes*) ... 650
13.2.4. Componentele şi pachetele bibliotecii Swing ... 652
13.2.5. Principii de bază ... 659
13.2.5.1. *MVC (*Model-View-Controller*)* ... 659
13.2.5.2. UI Delegate ... 660
13.2.5.3. Look-and-feel ... 661
13.2.5.4. *Mai multe despre modele* ... 664
13.2.5.5. *Swing* versus *AWT* ... 667

13.3. Fundamentele Swing ... 668
13.3.1. Evenimente şi ascultători ... 668
13.3.2. Fire de execuţie în Swing ... 681
13.3.3. Desenarea componentelor grafice ... 684
13.3.4. Alinierea la JavaBeans ... 689
13.3.5. Gestiunea „focusului" ... 691
13.3.6. Clasificarea componentelor Swing ... 695

13.4. Containere de bază ... 695
13.4.1. `JFrame` ... 696
13.4.2. Clase şi interfeţe legate de ferestre ... 701
13.4.3. `JWindow` ... 702
13.4.4. `JDialog` ... 705
13.4.5. `JApplet` ... 707

13.5. Containere intermediare ... 708
13.5.1. `JPanel` ... 708

13.5.2. `JScrollPane` 709
13.5.3. `JSplitPane` 713
13.5.4. `JTabbedPane` 715

13.6. Componente atomice simple 720
13.6.1. `JLabel` 720
13.6.2. Butoane 722
13.6.2.1. *`JButton`* 724
13.6.2.2. *`JToggleButton`*, *`JCheckButton`*, *`JRadioButton`* 724
13.6.3. Chenare 727
13.6.4. `JList` 730
13.6.5. `JComboBox` 734
13.6.6. `JSpinner` 738

13.7. Componente atomice complexe 739
13.7.1. Componente text 739
13.7.2. `JTable` 748
13.7.3. `JTree` 760

13.8. Meniuri şi bare de unelte 767
13.8.1. Meniuri 767
13.8.2. `JToolBar` 769

13.9. Dialoguri 774
13.9.1. `JOptionPane` 775
13.9.2. `JFileChooser` 780
13.9.3. `JColorChooser` 785

13.10. Componente pentru progres şi derulare 789
13.10.1. `JSlider` 789
13.10.2. `JScrollBar` 791
13.10.3. `JProgressBar` 793
13.10.4. `JToolTip` 795

13.11. Ferestre interioare 796
13.11.1. `JDesktopPane` 797
13.11.2. `JInternalFrame` 797

13.12. Recomandări şi optimizări 802

13.13. Concluzii 803

13.14. Test-grilă 804

13.15. Exerciţii propuse spre implementare 809

13.16. Proiecte propuse spre implementare 810

14. SISTEMUL HELP PENTRU JAVA 811

14.1. Cuvinte-cheie 811

14.2. Convenţii 812

14.3. Principii de bază 812
14.3.1. Prezentarea pachetului JavaHelp 813
14.3.2. Alcătuirea ferestrei standard JavaHelp 814
14.3.3. Principii pe care sistemul JavaHelp le respectă 815

14.4. Dezvoltarea efectivă a help-ului 815
14.4.1. Temele HTML 816
14.4.2. Fişierul HelpSet 817

- 14.4.3. Fişierul Map 820
- 14.4.4. Fişierul cuprins 821
- 14.4.5. Fişierul index 823
- 14.4.6. Indexarea şi căutarea 824
- 14.4.7. Compresia şi încapsularea 827

14.5. Ataşarea help-ului aplicaţiilor 827
- 14.5.1. Clasa `HelpSet` 828
- 14.5.2. Clasa `HelpBroker` 829
- 14.5.3. Clasa `CSH` 831
- 14.5.4. Adăugarea help-ului ferestrelor 832
- 14.5.5. Adăugarea help-ului componentelor 833
- 14.5.6. Activarea help-ului prin apăsarea butoanelor 833
- 14.5.7. Help activat folosind mouse-ul 834
- 14.5.8. Scufundarea help-ului în aplicaţie 835

14.6. Concluzii 836

14.7. Test-grilă 836

14.8. Exerciţii propuse spre implementare 837

14.9. Proiecte propuse spre implementare 837

15. INTERNAŢIONALIZAREA APLICAŢIILOR 839

15.1. Cuvinte-cheie 839

15.2. Introducere 840

15.3. Codări 841
- 15.3.1. Despre Unicode 841
- 15.3.2. Seturi de caractere 842

15.4. Setarea localizării 845

15.5. Separarea datelor 848

15.6. Formatări 851
- 15.6.1. Formatarea numerelor 852
- 15.6.2. Reprezentarea monetară 852
- 15.6.3. Formatarea datei calendaristice şi a orei 853
- 15.6.4. Formatarea mesajelor 853

15.7. Lucrul cu fonturi 855

15.8. Introducerea textului 856

15.9. Paşii necesari pentru internaţionalizarea aplicaţiilor 857

15.10. Concluzii 858

15.11. Test-grilă 858

15.12. Exerciţii propuse spre implementare 860

15.13. Proiecte propuse spre implementare 860

Bibliografie 861

Cuvânt înainte

Java este un limbaj de programare orientat obiect consacrat. Cele mai multe aplicaţii distribuite sunt scrise în Java, iar noile evoluţii tehnologice permit utilizarea sa şi pe dispozitive mobile gen telefon, agendă electronică, palmtop etc. În felul acesta se creează o platformă unică, la nivelul programatorului, deasupra unui mediu eterogen extrem de diversificat. Avantajele sunt evidente, atât pentru proiectanţi, care „scriu o dată şi execută pe orice maşină virtuală Java (JVM)", cât şi pentru utilizatori, care vor beneficia de un spectru îmbogăţit de servicii. De aceea se poate afirma cu încredere că Java este un câştigător, în lumea volatilă a tehnologiilor de calcul.

Cartea de faţă este un instrument foarte util pentru cei care doresc să înveţe Java şi să scrie aplicaţii distribuite sau, cu alt termen, aplicaţii în reţea. Cu entuziasm şi dedicare, cei trei autori au cuprins în opera lor, într-un stil facil şi plăcut, aspectele caracteristice ale limbajului Java, cât şi ale tehnologiilor care îl însoţesc şi permit scrierea de aplicaţii distribuite. Fiecare capitol are multe exemple de programare, bine documentate, şi se încheie cu întrebări, exerciţii şi proiecte propuse, care au rolul de a fixa cunoştinţele asimilate. În felul acesta, învăţarea limbajului de programare Java devine un exerciţiu practic, reconfortant.

Remarc cu plăcere că activitatea grupului de tineri de la Facultatea de Informatică din Iaşi, care studiază tehnologiile Web şi de calcul distribuit, se îmbogăţeşte atât calitativ, cât şi ca producţie de carte de specialitate. Este un aspect pozitiv, cu influenţe importante asupra procesului didactic.

prof. dr. ing. Dan Grigoraş
Computer Science Department
University College, Cork, Ireland

Facultatea de Informatică
Universitatea „Al.I. Cuza", Iaşi, România
6 februarie 2003

Prefaţă la ediţia a II-a

Java este un limbaj de programare de nivel înalt, orientat obiect, proiectat iniţial în scopul realizării de aplicaţii pentru internet şi, mai cu seamă, pentru Web. Acesta este utilizat cu succes în prezent şi pentru programarea aplicaţiilor destinate intranet-urilor. În acest sens, multe firme recurg la limbajul Java în procesul de informatizare întrucât oferă un foarte bun suport pentru tehnologiile de vârf şi, nu în ultimul rând, pentru că este gratuit şi în mod continuu îmbogăţit şi îmbunătăţit.

Cititorului i se oferă un ghid complet, cartea de faţă cuprinzând atât noţiunile de bază ale limbajului Java, cât şi elemente de nivel mediu şi avansat. De asemenea, sunt prezentate noţiuni de programare orientată obiect şi algoritmică, elemente de programare concurentă şi distribuită, toate acestea pentru ca limbajul Java să fie mai uşor de înţeles. De aceea, cartea poate fi un instrument util pentru însuşirea şi consolidarea cunoştinţelor pentru orice elev, student, programator, şi nu numai.

Primele patru capitole cuprind noţiuni de bază, de la instalare, compilare şi execuţie la tipuri de date şi instrucţiuni. Aceste prime capitole au rolul de a iniţia cititorul în limbajul de programare Java, aducându-l la un nivel mediu. Celelalte capitole prezintă treptat tehnologiile Java utile în realizarea de către programatori a aplicaţiilor, de la nivelul mediu până la avansat.

Cartea este bogată în exemple de programe Java scurte şi comentate pentru a ilustra cât mai bine cele prezentate. De regulă, fiecare capitol se termină cu prezentarea unei aplicaţii concrete pentru a realiza trecerea cititorului de la un cunoscător al limbajului Java la un programator Java. Toate capitolele au o structură de prezentare asemănătoare, începând cu secţiunea de cuvinte-cheie, apoi conţinutul efectiv, urmate de concluzii, test-grilă, exerciţii şi proiecte propuse spre implementare. Secţiunea de test-grilă reprezintă un suport pentru obţinerea de certificate internaţionale Java (de exemplu, *http://suned.sun.com/US/certification*, *http://studyguides.cramsession.com/cramsession/sun/java2*).

Faţă de prima ediţie au fost adăugate elementele noi din versiunile recent apărute (*e.g.* tipuri generice, enumerări, noile conversii implicite), au fost eliminate cele învechite, iar cele care s-au păstrat au fost revizuite. De asemenea, au fost adăugate exemple noi.

Datorită complexităţii şi spaţiului necesar pentru o tratare corespunzătoare, am optat să prezentăm mediul de dezvoltare *Eclipse*, documentarea, testarea automată şi depanarea aplicaţiilor Java într-o lucrare viitoare.

Pentru a putea stabili o legătură cu cei interesaţi de limbajul Java şi pentru a furniza informaţii suplimentare, cartea va avea la adresa *http://www.infoiasi.ro/~stanasa/java* o pagină de prezentare, conţinând rezolvarea testelor-grilă şi, eventual, a exerciţiilor propuse, alte exemple de programe Java, prezentarea unor unelte de programare Java etc.

Pentru suportul acordat realizării acestei cărţi mulţumim domnilor dr. Sabin Corneliu Buraga, Cristian Frăsinaru, Nicolae Tăbuşcă, Laurenţiu Constantin şi Bogdan Danu.

De asemenea, mulţumim familiilor noastre pentru tot sprijinul acordat.

Autorii
martie 2007

1. Introducere

1.1. Cuvinte-cheie

- limbaj orientat obiect, compilator, interpretor, platformă
- mediu de execuţie, mediu de dezvoltare
- instalare, Windows, Linux
- internet, Web, HTML, applet, servlet
- Java, C, C+ +, maşină virtuală Java

1.2. Istoricul şi caracteristicile limbajului Java

Java este un limbaj de programare orientat obiect, destinat în principal programării internet.

Începutul limbajului Java este în toamna anului 1991, când firma Sun Microsystems a finanţat un proiect cu numele *Green* condus de James Gosling. Scopul echipei *Green* era să plaseze firma Sun Microsystems pe piaţa produselor electronice comerciale. Inginerii şi dezvoltatorii de software de la Sun au căutat microprocesoare care să ruleze pe o multitudine de maşini, în particular pe sisteme distribuite care lucrează în timp real. Cheia succesului firmei Sun a fost abilitatea lucrului pe platforme multiple. După patru ani de lucru, echipa Green finalizează specificaţiile limbajului Java. Apoi, compania Sun Microsystems vinde licenţa firmelor IBM, Microsoft, Silicon Graphics, Adobe şi Netscape. Acum, Java e disponibil sub auspicii *open source.*

Caracteristicile limbajului Java sunt:

- limbaj *compilat şi interpretat*. Un limbaj este *compilat* dacă un program scris în acel limbaj este tradus într-un cod pe care calculatorul îl poate „înţelege" (executa) mult mai uşor. Un limbaj este *interpretat* dacă instrucţiunile unui program scris în acel limbaj sunt procesate linie cu linie. Programele interpretate sunt mai lente decât cele compilate, însă cele compilate sunt, de obicei, dependente de platforma respectivă. Programele Java sunt mai întâi compilate în nişte fişiere intermediare asemănătoare codului de asamblare (numite *byte code*, engl.), apoi acestea sunt interpretate de mediul de execuţie Java în instrucţiuni-maşină asociate platformei sistem.
- *limbaj independent de platformă*. La instalarea limbajului Java, se va crea o maşină virtuală Java care are drept scop traducerea instrucţiunilor unui *byte code* Java în instrucţiuni-maşină pentru platforma curentă. Astfel fişierele intermediare *byte code* pot fi copiate şi executate pe orice platformă (indiferent dacă este Windows, Unix etc.). Bineînţeles că nu trebuie să utilizăm elemente specifice unei platforme, cum ar fi delimitatorul utilizat în calea directoarelor: / pentru Unix, respectiv \ pentru Windows (se vor utiliza în schimb constantele `File.separatorChar` sau `File.separator`).
- *limbaj orientat obiect*. Cea mai importantă proprietate a limbajului Java este orientarea obiect, aceasta fiind privită ca o trăsătură implicită. Java pune în evidenţă toate aspectele legate de programarea orientată obiect: obiecte, trimitere de parametri, încapsulare, clase, biblioteci, moştenire şi modificatori de acces.
- *limbaj concurent*. Concurenţa (engl. *multithreading*) înseamnă capacitatea unui program de a executa mai multe secvenţe de cod în acelaşi timp. O secvenţă de cod Java se numeşte *fir de execuţie* (engl. *thread*). Datorită posibilităţii creării mai multor fire de execuţie, un program Java poate să execute mai multe sarcini simultan, de exemplu, animaţia unei imagini, transmiterea unei melodii spre placa de sunet, comunicarea cu un server, alocarea şi eliberarea memoriei etc.).
- *limbaj simplu*. Spre deosebire de C++, care este cel mai popular limbaj orientat obiect, Java elimină câteva dintre trăsăturile acestuia: posibilitatea moştenirii

multiple este exclusă, şirurile sunt încapsulate într-o structură clasă. Spre deosebire de C/C++, în Java nu există pointeri, iar alocarea şi dealocarea memoriei se fac automat. Există un mecanism de eliberare a memoriei (engl. *garbage collection*) pus în practică de un fir de execuţie de prioritate mică.

- *limbaj distribuit.* Java este distribuit, deoarece permite utilizarea obiectelor locale şi de la distanţă. Limbajul Java oferă posibilitatea dezvoltării de aplicaţii pentru internet, capabile să ruleze pe platforme distribuite şi eterogene. În acest sens, Java respectă standardul IEEE (engl. *Institute of Electrical and Electronics Engineers*) pentru structurile de date, cum ar fi folosirea întregilor, a numerelor în virgulă flotantă şi a şirurilor de caractere. Java se poate utiliza în aplicaţii de reţea, deoarece respectă protocoalele de reţea, cum ar fi FTP, HTTP, SOAP etc.
- *limbaj performant.* Interpretorul Java este capabil să execute un *byte code* aproape la fel de repede ca pe un cod compilat. Având posibilitatea să lucreze cu fire de execuţie multiple, Java justifică faptul că este un limbaj performant. De exemplu, un program Java poate aştepta citirea unor date, în timp ce un alt fir de execuţie poate aloca sau elibera memoria necesară programului.
- *limbaj dinamic şi robust.* Java întârzie alocarea obiectelor şi legarea dinamică a claselor până la momentul execuţiei. Astfel, se vor evita erorile de alocare chiar dacă mediul s-a schimbat de la ultima compilare a programului. Faptul că Java este robust se justifică prin eliminarea utilizării pointerilor, care era generatoare de erori în cazul programelor C/C++. În schimb, Java verifică memoria dinamic înainte de a fi alocată şi are un sistem automat de alocare/dealocare a memoriei.
- *limbaj sigur.* Programele Java nu pot accesa memoria *heap*, *stack* sau alte secţiuni protejate de memorie, deoarece Java nu foloseşte pointeri şi alocă memorie doar la execuţie. Înainte ca interpretorul Java să execute *byte code*-ul, se verifică dacă este un cod Java valid prin cercetarea accesului la date, a conversiilor de date nepermise, a valorilor şi parametrilor incorecţi, a depăşirii stivei.

1.3. Instalare

Limbajul Java se poate instala pe calculatoare cu diverse sisteme de operare, cum ar fi Windows, Linux, staţii Sun Solaris. Acest lucru este posibil, deoarece un program Java nu se execută direct de către microprocesor, ci utilizând un calculator ipotetic intermediar numit *maşina virtuală Java* (engl. *Java Virtual Machine – JVM*). Un compilator Java translatează codul-sursă Java într-un limbaj numit *byte code* (care e format din instrucţiuni-maşină pentru maşina virtuală Java) asemănător limbajului de asamblare şi pe care îl vom denumi în paginile acestei cărţi *cod binar Java*. Pentru execuţia unui program Java, interpretorul Java inspectează şi decodifică codul binar Java, verifică dacă este sigur de executat şi apoi îl execută instrucţiune cu instrucţiune. La instalarea limbajului Java, se creează, de fapt, corespondenţa dintre acţiunile codului binar Java şi instrucţiunile maşinii fizice. Acesta este „secretul" independenţei de platformă a limbajului Java.

Faptul că în procesul de traducere a codului-sursă Java în instrucţiunile maşinii fizice s-a interpus maşina virtuală Java implică o scădere a vitezei de execuţie faţă de un program echivalent care foloseşte instrucţiuni-maşină native. Dacă la începuturile limbajului Java un program era de 10 ori mai lent, în versiunile actuale ale pachetului JDK (engl. *Java Development Kit*) performanţele au crescut semnificativ, raportul vitezei de execuţie dintre un program Java şi un program scris în cod nativ fiind de aproximativ 2. Mai mult, pentru un mediu Java care suportă *compilarea în timpul încărcării* (engl. *Just-In-Time compilation*) nu va exista nici o diferenţă de viteză a execuţiei. Compilatoarele *Just-In-Time* vor traduce programele Java în instrucţiuni--maşină native chiar din momentul încărcării.

Sunt disponibile mai multe medii de dezvoltare a aplicaţiilor Java (JDK – *Java Development Kit*), realizate de diverşi producători printre care Sun, IBM, Borland. Firma Sun pune la dispoziţie gratuit kitul de instalare pentru sistemele de operare Windows, Linux şi Unix pe situl *http://java.sun.com/*. O versiune a kitului de instalare pentru sistemul de operare MacOS este disponibilă pe situl *http://devworld.apple.com/java/*.

Pentru crearea şi editarea programelor Java se poate utiliza orice editor obişnuit de texte, însă este recomandată utilizarea unui mediu specializat întrucât acestea reduc semnificativ timpul de dezvoltare a aplicaţiilor. Cele mai importante medii de programare pentru Java sunt Eclipse – *http://www.eclipse.org/*, NetBeans – *http://www.netbeans.org/* şi IntelliJ Idea – *http://www.jetbrains.com/idea/*.

Instrucţiuni detaliate despre modul de instalare a mediului Java se găsesc pe siturile care oferă kiturile de instalare.

Pentru sistemul de operare Windows, instalarea kitului JDK 1.6.0 se realizează implicit în directorul `C:\Program Files\Java\jdk1.6.0\` care va conţine subdirectoarele `bin`, `demo`, `include`, `jre`, `lib` şi `sample`. În subdirectorul `bin` se găsesc principalele instrumente ale pachetului JDK:

- `javac.exe` – compilator Java;
- `java.exe` – interpretor Java;
- `appletviewer.exe` – instrument de vizualizat appleturi;
- `javadoc.exe` – generator de documetanţii;
- `jar.exe` – instrument pentru arhive jar;
- `javap.exe` – dezasamblor de fişiere *byte code*.

Apoi se poate dezarhiva şi documentaţia pentru care se va crea un subdirector `docs`. Directorul `C:\Program Files\Java\jdk1.6.0\bin` trebuie să fie prezent în variabila de mediu `PATH`. Codul-sursă al bibliotecilor claselor este inclus în arhiva `src.zip` din directorul în care s-a realizat instalarea. Fişierele-sursă se pot extrage folosind utilitarul jar astfel:

```
jar xvf src.zip
```

Dezarhivarea surselor nu este obligatorie. Există editoare performante de programe Java pentru care indicarea locaţiei arhivei în codul-sursă conduce la afişarea documentaţiei (conţinutul comentariilor *javadoc*). În Eclipse, aceasta se realizează prin intermediul *tooltip*-urilor.

Pentru sistemul Linux va trebui să procurăm un fişier cu extensia `.bin`, de exemplu, pentru JDK 1.6.0 există `jdk-1_6_0-linux-i586-rpm.bin`. Acesta va fi executat de administratorul sistemului (utilizatorul *root*) astfel:

```
./jdk-1_6_0-linux-i586-rpm.bin
```

Vor fi afişaţi termenii de utilizare a mediului Java şi, în cazul în care suntem de acord, vom tasta cuvântul `yes`. La execuţia acestui program va rezulta fişierul `jdk-1_6_0-linux-i586.rpm`, cu care vom realiza instalarea efectivă:

```
rpm -i jdk-1_6_0-linux-i586-rpm
```

Pachetul JDK va fi instalat în directorul `/usr/java/jdk1.6.0/`, iar pentru a putea utiliza mai uşor comenzile puse la dispoziţie vom edita fişierul `.bash_profile` din contul personal, adăugând următoarea linie:

```
PATH=/usr/java/jdk1.6.0/bin:$HOME/bin:$PATH
```

Astfel, directorul care conţine comenzile din mediul Java a fost inclus în calea de căutare a fişierelor executabile. Structura de directoare este similară cu cea din sistemul Windows, diferenţa fiind la numele comenzilor, acestea neavând extensie (în Windows ele au extensia `.exe`). De asemenea, comenzile sunt utilizate cu aceiaşi parametri ca pentru sistemul Windows.

1.4. Tipuri de aplicaţii Java

În principiu, există mai multe tipuri de aplicaţii Java:

a) *aplicaţii de sine stătătoare* (engl. *stand-alone*);
b) *aplicaţii care se execută pe partea de client* (numite appleturi);
c) *aplicaţii care se execută pe partea de server* (numite servleturi).

a) O aplicaţie de sine stătătoare Java este un program care poate fi executat în afara contextului unui browser (navigator) Web. Principala caracteristică a acestui tip de aplicaţie este încapsularea în cadrul clasei principale a unei funcţii `main()`, asemănătoare cu cea folosită în C, care are următoarea signatură:

```
public static void main(String [] args)
```

Aplicaţiile Java de sine stătătoare se pot împărţi la rândul lor în:

- aplicaţii de consolă, care rulează la linia de comandă; de exemplu, fereastră de consolă (engl. *command prompt*) pe un calculator personal rulând Windows;
- aplicaţii-fereastră, care lucrează cu ferestre grafice folosind interfaţa grafică cu utilizatorul (engl. *Graphical User Interface* – GUI).

În continuare vom edita un text-sursă pe care îl vom salva în fişierul `AplicatiaUnu.java` ca aplicaţie de consolă.

```
public class AplicatiaUnu {
```

```
public static void main (String args[]) {
if (args.length == 0)
      System.out.println("Aplicatie independenta.");
else {
      System.out.println("Argumentele programului sunt: ");
      for (int i = 0; i < args.length; i++)
       System.out.println(args[i]);
    }
  }
}
```

Chiar dacă este un program simplu, acesta înglobează multe dintre trăsăturile unui program orientat obiect complex, şi anume: are un obiect (implicit generat de sistemul Java), utilizează încapsularea şi transmiterea parametrilor, defineşte o clasă, foloseşte moştenirea şi modificatori de acces. Compilarea acestui program Java se realizează cu comanda:

```
javac AplicatiaUnu.java
```

Astfel, se va crea fişierul în cod binar Java cu numele `AplicatiaUnu.class`. Execuţia acestui program se va realiza folosind în mod similar comanda:

```
java AplicatiaUnu
```

Pe ecran va apărea mesajul:

```
Aplicatie independenta.
```

Este de remarcat faptul că la compilare trebuie scrisă şi extensia fişierului (`.java`), pe când la interpretare, aceasta nu trebuie specificată.

Continuăm cu câteva explicaţii asupra codului Java de mai sus. Fişierul constă în definiţia unei clase declarate `public` intitulate `AplicatiaUnu` (de altfel, aceasta este o cerinţă obligatorie: numele clasei declarate `public` coincide cu numele fişierului Java). Metoda `main()` este punctul normal de intrare pentru o aplicaţie Java. Aceasta este declarată `public` prin convenţie. Însă trebuie declarată obligatoriu `static` pentru a putea fi executată fără a construi o instanţă a clasei corespunzătoare. Şirul `args[]` conţine argumentele pe care utilizatorul poate să le specifice la linia de comandă. De exemplu, dacă executăm programul `AplicatiaUnu` astfel:

```
java AplicatiaUnu Aceasta este o aplicatie independenta
```

obţinem `args[0]` egal cu „`Aceasta`" şi `args[1]` egal cu „`este`", şi tot aşa până la `args[4]`, care va fi „`independenta`". Atenţie! Ca şi în C, şirul începe de la locaţia de index 0 şi se termină la locaţia de index 4 (lungimea şirului `args` fiind 5). Numele clasei (`AplicatiaUnu`) şi numele comenzii (`java`) nu apar în şirul `args` (cum se întâmplă în limbajul C, de exemplu). Numele dat acestui argument (`args`) este orientativ şi poate fi schimbat după dorinţă.

Clasa `System` conţine câmpul:

```
public static final PrintStream out;
```

care reprezintă fluxul de ieşire standard. Acest flux este deja deschis şi pregătit să accepte afişarea datelor.

În clasa `PrintStream` există metoda:

```
public void println(String x);
```

care are drept scop afişarea şirului `x` şi terminarea liniei. Astfel, linia:

```
System.out.println("Aplicatie independenta.");
```

va avea drept scop afişarea la consolă a mesajului „`Aplicatie independenta`".

b) Aplicaţiile care se execută pe partea de client sunt cele încărcate de pe un server (de cele mai multe ori aflat la distanţă) şi apoi executate de programe speciale cum ar fi navigatoarele Web. De exemplu, appleturile sunt aplicaţii care sunt executate de către navigatorul Web, iar middleturile sunt executate de către dispozitive mobile fără fir (mai exact de către sistemul software al acestora).

Atenţie! Pentru ca aceste aplicaţii să poată fi executate trebuie să fie instalat mediul de execuţie Java!

Un applet este un program Java respectând o mulţime de convenţii care îi permit să ruleze în cadrul unui navigator Web ce încorporează o maşină virtuală Java. Tehnologia appleturilor este similară, într-un anumit sens, cu a programelor JavaScript. Fişierul cu extensia `.class` al unui applet este stocat pe un server Web şi poate fi accesat de către un client prin intermediul unei pagini care conţine acel applet. La încărcarea paginii, codul appletului este transferat pe maşina-client şi va fi executat cu ajutorul maşinii virtuale incluse în navigator.

Atenţie! Navigatorul Web trebuie să ofere suport pentru rularea appleturilor!

Fie textul-sursă Java memorat în fişierul `AplicatiaDoi.java`:

```
import java.awt.Graphics;
import java.applet.Applet;

public class AplicatiaDoi extends Applet {
  public void paint(Graphics g) {
        g.drawString("Acesta este un applet", 50, 50);
  }
}
```

Compilarea acestui program Java se realizează astfel:

```
javac AplicatiaDoi.java
```

În acest mod se va crea fişierul *byte code* cu numele `AplicatiaDoi.class`. Vom scrie apoi un fişier HTML, intitulat `AplicatiaDoi.html`, care va avea inclus marcatorul `<applet>`. De exemplu, ar putea fi:

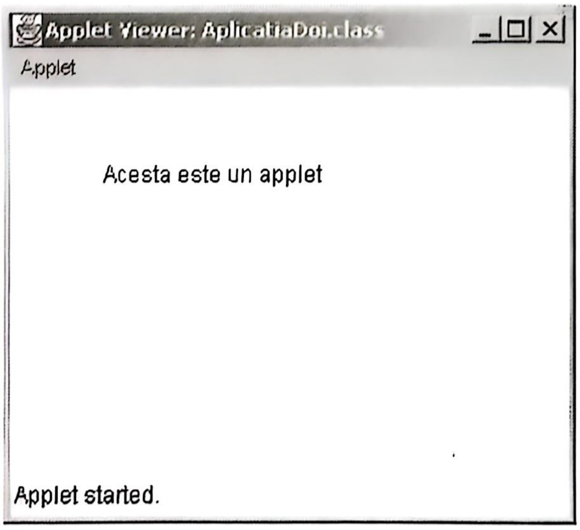
```
<html>
<head>
  <title>Exemplu de applet</title>
</head>

<body>
  <applet code="AplicatiaDoi.class" width="300" height="200">
  </applet>
</body>
</html>
```

În versiunea HTML 4.0, marcatorul `<applet>` este considerat învechit (engl. *deprecate*). Ca alternativă se foloseşte marcatorul `<object>`. De exemplu:

```
<html>
<head>
  <title>Exemplu de applet folosind elementul obiect</title>
</head>

<body>
  <object codetype="application/java"
   classid="java:AplicatiaDoi.class"
   width="300" height="200">
  </object>
</body>
</html>
```

Execuţia acestui program se va realiza prin intermediul unui navigator Web (de exemplu, Firefox sau Microsoft Internet Explorer), care va apela fişierul `AplicatiaDoi.html`, sau rulând un vizualizator de appleturi cu comanda:

```
appletviewer AplicatiaDoi.html
```



Acesta este un applet

Applet started.

Rularea unui applet cu `appletviewer`

Continuăm cu câteva explicaţii referitoare la codul Java de mai sus. Fişierul constă în definiţia unei clase declarate publică, intitulate `AplicatiaDoi`. Mai întâi, avem două instrucţiuni de import al unor clase. Este vorba de clasa `Graphics` din pachetul `java.awt` (Abstract Windows Toolkit) şi clasa `Applet` din pachetul `java.applet`. Urmează apoi declaraţia clasei `AplicatiaDoi`. Prezenţa cuvântului rezervat `extends` semnifică derivarea clasei `AplicatiaDoi` din clasa `Applet`. Urmează definiţia metodei `paint()` care aparţine clasei `Component`. Clasa `Component` este situată cu trei niveluri mai sus decât clasa `Applet`, adică ierarhia claselor este: `Object`, `Component`, `Container`, `Panel`, `Applet`. Metoda `paint()` este automat apelată când appletul decide că este nevoie de actualizarea sa, de exemplu, când appletul este plasat prima dată pe ecran, când s-a mutat appletul sau când altă fereastră a fost deasupra appletului. De fapt, appletul apelează metoda `update()` din aceeaşi clasă `Component`, care la rândul ei apelează metoda `paint()`. Sintaxa metodei `paint()` este:

```
public void paint(Graphics g);
```

Obiectul `g` din clasa `Graphics` reprezintă suprafaţa pe care se poate desena, în cazul unui applet fiind aria din interiorul appletului. Clasa `Graphics` are foarte multe metode, însă cea mai folosită metodă pentru afişarea unui şir de caractere este `drawString()`. Această metodă are sintaxa:

```
public abstract void drawString(String str, int x, int y);
```

Apelul acestei metode asupra unui obiect din clasa `Graphics` are ca efect afişarea şirului `str` începând de la poziţia `(x,y)` a appletului. Colţul din stânga-sus al appletului are coordonata O(0, 0), iar pe orizontală avem semiaxa Ox şi pe verticală Oy. Dimensiunea appletului se fixează folosind atributele `width` (lăţime = Ox) şi `height` (înălţime = Oy) ale marcatorului `<applet>` din fişierul HTML care va apela fişierul *byte code* asociat programului Java. Astfel, execuţia programului `AplicatiaDoi` va consta în afişarea textului „`Acesta este un applet`" începând cu punctul de coordonate (50, 50) al appletului de dimensiuni (300, 200).

De la apariţia sa şi până în prezent, Java a fost îmbunătăţit continuu, transformându-se într-un mediu de dezvoltare pentru o largă varietate de aplicaţii, cum ar fi aplicaţii de tip client/server, aplicaţii internet sau aplicaţiile standard independente.

Cea mai recentă platformă Java este J2ME (Java 2 Micro Edition), ce oferă suportul pentru crearea aplicaţiilor care rulează pe dispozitive mici fără fir (engl. *wireless devices*), cele mai cunoscute la noi fiind telefonul mobil şi pagerul.

J2ME face parte din platforma Java 2 care mai cuprinde J2EE (Java 2 Enterprise Edition) şi J2SE (Java 2 Standard Edition).

Tehnologia J2ME este asemănătoare cu cea J2SE, diferenţele constând în faptul că dispozitivele pentru care sunt destinate aplicaţiile J2ME folosesc resurse limitate. Pentru a crea astfel de aplicaţii este folosit un subset de clase din pachetele `java.lang`, `java.io`, `java.util`, pachete care fac parte din J2SE. În plus, pentru J2ME au fost construite două maşini virtuale: C Virtual Machine (CVM) şi K Virtual Machine (KVM). Acestea ocupă puţină memorie şi necesită mult mai puţine resurse decât maşina virtuală pentru tehnologiile J2EE sau J2SE.

Pentru a rula KVM, un dispozitiv fără fir trebuie să aibă anumite caracteristici hardware. Printre acestea se numără condiţiile minimale:

- peste 160 KB (de obicei, minim 512 KB) de memorie internă pentru a rula platforma Java;
- procesor pe 16 biţi sau 32 biţi cu viteza de 25 MHz;
- consum redus de energie oferit adesea de o baterie;
- conectare la un anumit tip de reţea, adesea fără fir (engl. *wireless network*), cu o lărgime de bandă limitată la 9600 bps;
- rezoluţia de 96x54;
- adâncimea de culoare de 1 bit (adică alb/negru).

Mediul J2ME este furnizat de către Sun în cadrul pachetului Java 2 Micro Edition Wireless Toolkit (J2MEWTK). Acest pachet este disponibil pe două platforme: Microsoft Windows şi Sun Solaris.

O aplicaţie care poate rula pe un dispozitiv „fără fir" se numeşte middlet. Un astfel de middlet se aseamănă ca tehnologie cu cea a appleturilor Java.

c) O aplicaţie ce se execută pe partea de server este o aplicaţie care este rulată de către un server ca urmare a unei cereri primite de acesta, iar rezultatul este trimis programului solicitant. De cele mai multe ori acestea extind funcţionalitatea respectivului server. De exemplu, servleturile sunt aplicaţii care se execută pe partea de server.

Un servlet este o componentă Web, scrisă în Java, care poate fi încărcată dinamic şi rulată de un server Web, şi care poate interacţiona cu diferiţi clienţi folosind o implementare a paradigmei cerere/răspuns bazată pe protocolul HTTP. Servleturile extind funcţionalitatea unui server, de obicei, a unui server HTTP. Spre deosebire de appleturi, servleturile nu afişează o interfaţă grafică către client, ci doar rezultatul procesării (de obicei, sub forma unui document HTML).

Un caz particular de servleturi îl constituie JSP-urile (*Java Server Pages*). Această tehnologie este atât o extensie a servleturilor, cât şi o extensie a limbajului HTML. Prin intermediul unor taguri particulare se pot insera declaraţii, expresii, secvenţe de cod Java etc.

Unele aplicaţii mai complexe au nevoie de o cantitate mare de informaţii structurate, iar pentru acest lucru se recurge la utilizarea bazelor de date. Conexiunile la baze de date se realizează în Java prin intermediul JDBC (*Java Database Connection*).

JDBC este o specificare API (*Application Programming Interface*) dezvoltată de Sun care defineşte o interfaţă uniformă pentru accesarea diferitelor baze de date. JDBC este parte a platformei Java şi este inclusă în pachetul JDK. Cea mai importantă funcţie a JDBC-ului este posibilitatea lucrului cu instrucţiuni SQL (engl. *Structured Query Language*) şi procesarea rezultatelor într-o manieră independentă şi consistentă de bazele de date. JDBC furnizează acces orientat pe obiecte la bazele de date prin definirea de clase şi interfeţe care reprezintă obiecte, cum ar fi conexiuni la baze de date, instrucţiuni SQL, mulţimi-rezultat, obiecte binare şi şiruri de caractere de dimensiuni mari, drivere de baze de date, manageri de drivere.

Aplicaţiile, fie de sine stătătoare, fie pe partea de client sau server, pot stabili conexiuni la baze de date, la diverse servere din reţea, pot salva starea obiectelor pentru a le restaura la o execuţie ulterioară sau pentru trimiterea în fluxuri de date. De asemenea, pot prelucra date în diferite formate: text (documente TXT, HTML, XML, RTF etc.), grafic (2D şi 3D) şi video.

În zilele noastre, cele mai multe aplicaţii sunt concepute pentru a rula în reţele de calculatoare, comunicând prin fluxuri de date (prin socketuri) sau apeluri de metode la distanţă. Informaţiile sunt transmise fie prin încapsulare în obiecte, fie prin mesaje XML. Utilizând formatul XML, comunicarea se poate realiza între aplicaţii care sunt implementate în limbaje de programare diferite.

Unele aplicaţii au nevoie de executarea metodelor altor obiecte aflate pe servere la distanţă. Aceste aplicaţii transmit parametrii metodei şi primesc rezultatul returnat de respectiva metodă.

Apelul metodelor la distanţă (engl. RMI – *Remote Method Invocation*) furnizează în nivelurile de reţea intermediare posibilitatea obiectelor Java, care se află pe maşini virtuale diferite, să comunice folosind apeluri de metode normale. Scopul implementării apelului metodelor la distanţă în Java este furnizarea unui cadru pentru comunicarea între obiecte prin metodele lor, indiferent de localizare. Aceasta înseamnă că un obiect client poate accesa un obiect având rol de server aflat pe o maşină la distanţă ca şi cum ar rula în aceeaşi maşină virtuală Java.

Pentru crearea unei clase care va fi accesibilă de la distanţă, trebuie să definim o interfaţă ale cărei metode sunt publice. Parametrii şi valorile returnate pot fi de orice tip; transferul de date este rezolvat automat de fluxul de obiecte. Clasa trebuie să implementeze această interfaţă (şi orice altă interfaţă şi metode necesare folosirii locale). Există un instrument disponibil sub JDK, numit `rmic` (engl. *remote method invocation compiler*), care generează două fişiere `.class` importante pentru realizarea apelurilor la distanţă.

1.5. Maşina virtuală Java

Maşina virtuală Java este noutatea principală a limbajului Java faţă de celelalte limbaje de programare. Partea dificilă a creării codului binar Java este compilarea codului-sursă pentru o maşină care nu există fizic. Această maşină este numită *maşina virtuală Java*, deoarece există doar în memorie. Apoi, interpretorul Java traduce instrucţiunile cod-maşină Java în mulţimi de instrucţiuni care pot fi înţelese de maşina reală (fizică).

Fără a intra în detalii de construcţie a maşinii virtuale, vom prezenta cele mai importante părţi ale unei astfel de maşini care necesită simularea unui calculator real: o mulţime de regiştri, o stivă, un mediu de execuţie, un ansamblu pentru eliberarea memoriei, o tabelă (engl. *pool*) de literali constanţi, un spaţiu de memorie a unei metode şi o mulţime de instrucţiuni.

Pentru examinarea instrucţiunilor *byte code*, JDK include un instrument software numit *dezasamblor de fişiere în cod binar Java*. Totuşi, cu acest utilitar nu se poate dezasambla un fişier `.class` pentru a crea fişierul-sursă Java. Dezasamblorul `javap` oferă doar posibilitatea de a vedea cum lucrează o clasă, cum foloseşte o clasă resursele sistem şi pentru a verifica instrucţiunile de importare a pachetelor şi dependenţele.

Dacă se execută `javap` fără opţiuni, atunci se va afişa numele fişierului `.class` compilat şi declaraţiile abreviate pentru variabilele, metodele şi clasele publice. Dacă se execută `javap` fără opţiuni şi fără argumente de tip clasă, atunci se va afişa lista tuturor opţiunilor.

Exemplul 1.5.1. Comanda:

```
javap -c -p -l -s -v AplicatiaUnu
```

va afişa:

```
Compiled from AplicatiaUnu.java
class AplicatiaUnu extends java.lang.Object {
    AplicatiaUnu();
     /*    ()V   */
     /* Stack=1, Locals=1, Args_size=1 */
    public static void main(java.lang.String[]);
     /*    ([Ljava/lang/String;)V    */
     /* Stack=2, Locals=1, Args_size=1 */
}

Method AplicatiaUnu()
   0 aload_0
   1 invokespecial #1 <Method java.lang.Object()>
   4 return

Line numbers for method AplicatiaUnu()
   line 1: 0

Method void main(java.lang.String[])
   0 getstatic #2 <Field java.io.PrintStream out>
   3 ldc #3 <String "Vom afisa acest mesaj.">
   5 invokevirtual #4 <Method void println(java.lang.String)>
   8 return

Line numbers for method void main(java.lang.String[])
   line 3: 0
   line 4: 8
```

Opţiunile de mai sus au semnificaţia următoare:

- c : pentru dezasamblarea codului;
- p : pentru afişarea tuturor membrilor şi claselor;
- l : pentru afişarea numerelor liniilor şi a tabelelor variabilelor locale;
- s : pentru tipărirea signaturii de tip interne;
- v : pentru afişarea mărimii stivei, precum şi a numărului de variabile locale şi argumente pentru metode.

Astfel, se observă că, atunci când compilăm un program Java, compilatorul traduce fiecare linie de cod de nivel înalt în mai multe linii de instrucţiuni de tip *byte code* (asemănătoare cu un limbaj de asamblare). În exemplul de mai sus, compilatorul inserează un constructor implicit, apoi metoda `main()`. De exemplu, instrucţiunile de asamblare de mai sus au următorul sens:

- `aload_0` : încarcă referinţa obiect dintr-o variabilă locală cu index specificat;
- `invokespecial` : apelează un constructor;
- `return` : metoda curentă returnează `void`;
- `getstatic` : obţine un câmp static din clasa curentă;
- `ldc` : memorează o constantă în tabela de constante din clasa curentă;
- `invokevirtual` : apelează o metodă în timpul execuţiei (şi nu în timpul compilării).

Detalii despre compilarea codului-sursă java în *byte code* se pot găsi în lucrarea [NoS96].

1.6. Concluzii

Kitul de instalare pentru mediul de dezvoltare Java se poate obţine în mod gratuit de la adresa *http://java.sun.com*. Maşina virtuală Java este noutatea principală a limbajului Java faţă de alte limbaje de programare orientate obiect. La instalarea limbajului Java, se va crea o maşină virtuală Java care are drept scop traducerea instrucţiunilor unui *byte code* în instrucţiuni-maşină pentru sistemul de operare curent. Astfel, fişierele intermediare *byte code* pot fi copiate şi executate pe orice platformă (indiferent dacă este Windows, Unix, sau altele).

În principiu, există trei tipuri de aplicaţii Java: *aplicaţii de sine stătătoare*, *aplicaţii care se execută pe partea de client* şi *aplicaţii care se execută pe partea de server*.

Aplicaţiile de sine stătătoare sunt programe Java care pot fi rulate în afara contextului unui navigator. Cea mai importantă caracteristică este încapsularea în clasa principală a unei funcţii `main()`, asemănătoare cu cea folosită în C.

Appleturile sunt încărcate de la server şi rulează pe maşina clientului. În general, appleturile se încapsulează într-o pagină Web şi afişează o interfaţă grafică spre client. Acestea se apelează cu un browser Web sau folosind comanda `appletviewer` pentru un fişier HTML 3.2 care conţine un marcator de forma `<applet>`, a cărui atribut *code* este egal cu numele appletului. În versiunea HTML 4.0, `<applet>` este considerat învechit, appleturile fiind adăugate în pagini cu ajutorul marcatorului `<object>`.

Un middlet este o aplicaţie care se poate rula pe dispozitive mobile fără fir, cum ar fi pagerele şi telefoanele mobile. Acestea au la dispoziţie resurse limitate.

Un servlet este o componentă Web care rulează pe un server Web şi care interacţionează cu diferiţi clienţi prin intermediul paradigmei cerere/răspuns a protocolului HTTP. Spre deosebire de appleturi, servleturile nu afişează o interfaţă grafică către client, ci doar rezultatul procesării servleturilor (de obicei, sub forma unui fişier HTML).

Aplicaţiile amintite pot avea conexiuni la bazele de date prin intermediul JDBC sau pot executa metode aflate la distanţă cu ajutorul tehnologiei RMI.

1.7. Test-grilă

Întrebarea 1.7.1. Este limbajul Java independent de platformă?
a) Da, deoarece se va crea o maşină virtuală Java care are drept scop traducerea instrucţiunilor unui *byte code* în instrucţiuni-maşină pentru sistemul de operare curent;
b) Nu, deoarece este imposibil ca acelaşi program Java să fie executat pe diferite sisteme de operare fără a fi modificat.

Întrebarea 1.7.2. Cum putem iniţializa pointeri în Java?
a) folosind operatorul de adresă &;
b) folosind operatorul de dereferenţiere *;
c) nu există pointeri în Java.

Întrebarea 1.7.3. Ce tipuri de aplicaţii Java există?
a) aplicaţii de sine stătătoare;
b) baze de date;
c) aplicaţii care se execută pe server sau client;
d) clienturi.

Întrebarea 1.7.4. Cum se poate executa appletul intitulat `UnApplet.class`?
a) folosind comanda `java UnApplet`
b) utilizând comanda `appletviewer` şi un fişier HTML care conţine un marcator de forma `<applet code="UnApplet.class" . . . >`
c) utilizând comanda `javac`

Întrebarea 1.7.5. Cu ce comandă putem dezasambla un cod binar Java?
a) `java -p`
b) `javac`
c) `javap`
d) `rmic`

1.8. Exerciţii propuse spre implementare

Exerciţiul 1.8.1. Instalaţi mediul de dezvoltare Java de la adresa *http://java.sun.com* şi rulaţi exemplele prezentate în secţiunile precedente.

2. Fundamentele limbajului Java

2.1. Cuvinte-cheie

- fişiere-sursă, pachete, definiţii import, definiţii de clase
- *case-sensitive*
- atomi lexicali (identificatori, cuvinte rezervate, literali, separatori, operatori, comentarii)
- tipuri de date (primitive, referinţă)
- variabile (date membre, locale)
- expresii, declaraţii (întregi, raţionali în virgulă flotantă, caracter, tablouri), conversii
- structuri de control (blocuri şi instrucţiuni, expresii condiţionale)

2.2. Fişiere-sursă

Toate fişierele-sursă Java trebuie să se termine cu extensia `.java`. Un fişier-sursă ar trebui să conţină cel mult o definiţie a unei clase de bază publice. Dacă există o clasă publică, atunci numele clasei trebuie să coincidă cu numele fişierului (bineînţeles, fără extensie). Un fişier-sursă poate conţine un număr nelimitat de definiţii de clase nepublice.

Există trei elemente de prim rang care pot apărea în fişier. Nici unul nu este obligatoriu, însă dacă apar, atunci trebuie scrise în ordinea: declaraţii de pachete, instrucţiuni de includere (engl. *import*) de clase, pachete şi/sau constante, apoi definiţii de clase.

Un *pachet* este o mulţime de clase Java (de obicei, sub forma fişierelor de cod binar Java) păstrate într-un acelaşi director. Există mai multe pachete de bază, cum ar fi: `java.lang`, `java.applet`, `java.io` etc. Pachetul `java.lang` nu necesită a fi declarat în programele Java, acesta fiind importat implicit. Ca urmare, putem avea programe Java care să nu importe explicit pachete.

Pachetele sunt utile pentru separarea claselor în vederea reutilizării lor. De obicei, aceste clase nu vor mai suferi modificări majore. Spre exemplu, un programator a implementat o serie de clase pentru lucrul cu numere fracţionare. Acestea pot fi stocate în acelaşi director şi vor putea fi reutilizate în alte aplicaţii prin includerea pachetului creat.

Formatul declarării unui pachet este simplu, necesitând scrierea cuvântului rezervat `package` urmat de numele pachetului. Numele pachetului este un şir de caractere format din cuvinte separate prin puncte. După ce s-au creat fişierele-clasă, acestea trebuie memorate într-o structură de subdirectoare care reflectă numele pachetelor. Pentru a păstra independenţa de platformă, Java înlocuieşte simbolurile „/" din Unix şi „\" din Windows cu caracterul „.".

Instrucţiunile de import de fişiere au o formă similară, putând importa o clasă dintr-un pachet sau un pachet întreg. Sintaxa generală în aceste cazuri este:

```
import <numePachet>.<numeClasa>;
```

respectiv:

```
import <numePachet>.*;
```

Exemplul 2.2.1. Pentru a exemplifica crearea şi importarea pachetelor, considerăm două fişiere-sursă plasate în directorul `C:\temp\pachet\exemplu\` de pe un sistem Windows care conţin câte o clasă simplă fiecare, astfel:

```
package pachet.exemplu;

public class ClasaUnu {
  public ClasaUnu() {
    System.out.println("Suntem in ClasaUnu");
  }
}
```

respectiv:

```
package pachet.exemplu;

public class ClasaDoi {
  public ClasaDoi() {
    System.out.println("Suntem in ClasaDoi");
  }
}
```

După compilarea acestor fişiere-sursă, vom crea un alt fişier-sursă care va importa clasele definite în pachetul `pachet.exemplu`. Acesta poate fi plasat oriunde în sistemul de fişiere. Conţinutul fişierului este:

```
import pachet.exemplu.*;

public class TestPachet {
  public static void main(String[] args) {
    ClasaUnu cu = new ClasaUnu();
    ClasaDoi cad = new ClasaDoi();
  }
}
```

La compilarea acestui fişier trebuie ca în variabila `CLASSPATH` să adăugăm şi directorul `C:\temp`. `CLASSPATH` este o variabilă de mediu ce conţine lista tuturor directoarelor în care se caută fişierele `.class` conţinând clase utilizate de aplicaţiile Java. Această variabilă este utilizată atât la compilare, cât şi la execuţie.

Dacă nu dorim să modificăm variabila de mediu `CLASSPATH`, putem indica la compilare unde se găsesc clasele necesare astfel:

```
javac -cp C:\Temp TestPachet.java
```

Pentru execuţie se procedează similar:

```
java -cp C:\Temp TestPachet
```

La execuţie se vor putea accesa clasele definite în pachetul `pachet.exemplu` şi rezultatul va fi:

```
Suntem in ClasaUnu
Suntem in ClasaDoi
```

Cel mai bine este să se specifice la compilare sau la execuţie locaţia claselor sau a bibliotecilor utilizate pentru a evita eventualele conflicte între clase cu versiuni diferite.

2.3. Atomi lexicali

Etapa de compilare a programelor Java începe cu *analizarea lexicală* a codului-sursă Java. Analizarea lexicală înseamnă parcurgerea secvenţială a textului identificând cuvinte acceptate de sintaxa Java, numite *atomi lexicali* (engl. *tokens*). Aceşti atomi lexicali sunt identificatori, cuvinte rezervate, literali, separatori şi operatori ai limbajului Java. Spaţiile şi comentariile sunt de asemenea identificate, însă acestea nu mai sunt transmise copiei programului Java în vederea continuării procesului de compilare.

2.3.1. Caractere Unicode

Programele Java pot fi scrise folosind setul de caractere ASCII şi eventual codări Unicode în notaţia `\udddd`. Astfel, traducerea lexicală poate folosi orice secvenţă Unicode cu sens special (engl. *escape*) pentru a include orice caracter Unicode folosind doar caractere ASCII (engl. *American Standard Code for Information Interchange*).

Detalii despre caracterele Unicode se găsesc la adresele:

- *http://www.unicode.org/unicode/standard/standard.html*
- *http://ietf.org/rfc/rfc2278.txt*

Limbajul Java este *case-sensitive*, adică face diferenţierea între caracterele mari şi mici. De exemplu, cuvântul `for` este diferit de `FOR` sau `For`.

În ceea ce priveşte reprezentarea textului şi codificarea caracterelor, Java utilizează două feluri de reprezentări text:

- setul de caractere Unicode – pentru reprezentarea internă a caracterelor şi a şirurilor de caractere;
- codări UTF – pentru intrări şi ieşiri.

Unicode foloseşte 16 biţi pentru reprezentarea unui caracter. Dacă cei mai semnificativi 9 biţi sunt toţi 0, atunci codificarea corespunde standardului ASCII (ultimii 7 biţi fiind reprezentarea caracterelor). Altfel, biţii reprezintă un caracter care nu este reprezentat în ASCII pe 7 biţi. Tipul `char` din Java corespunde unui caracter şi foloseşte codificarea Unicode. Clasa `String` conţine o înşiruire de caractere Java.

Setul de caractere Unicode este suficient pentru majoritatea limbilor, dar nu şi pentru limbile asiatice, care conţin prea multe caractere. Soluţia acestei probleme constă în utilizarea formatului UTF (engl. *Universal Character Set Transformation Format*). Codificarea UTF foloseşte exact numărul dorit de biţi: mai puţini pentru alfabete mici, mai mulţi pentru alfabete mari.

Prin *codificarea* caracterelor înţelegem o corespondenţă bijectivă între mulţimea caracterelor şi o mulţime de numere binare. Fiecare platformă Java are o codificare a caracterelor implicită, care este utilizată pentru interacţiunea dintre maşina Java care foloseşte Unicode intern şi sistemul de operare peste care este instalată aceasta. Codificarea implicită a caracterelor reflectă limba şi cultura locală. Fiecare codificare are un nume. De exemplu, ISO-8859-1 înseamnă Latin-1, ISO-8859-8 este ISO Latin/Hebrew şi Cp1258 este vietnameză.

Exemplul 2.3.1. Următorul program Java va returna numele codificării caracterelor de pe platforma Java curentă (de exemplu, Cp1250). Vom folosi metoda `getEncoding()` din clasa `InputStreamReader`.

```
import java.io.InputStreamReader;

public class CaractereUnicode {
  public static void main(String args[]) {
    InputStreamReader isr = new InputStreamReader(System.in);
    System.out.println(isr.getEncoding());
  }
}
```

Când are loc o operaţie de intrare/ieşire în Java, sistemul trebuie să cunoască codificarea caracterelor. Clasele I/O utilizează, de obicei, codificarea implicită locală, în caz că nu se precizează una explicită. Deşi codificarea implicită locală este adecvată, în aplicaţiile de reţea, când se comunică într-o reţea cu un alt computer, acestea trebuie să folosească aceeaşi codificare. În acest caz, este recomandată cererea explicită ISO-8859-1. Pentru mai multe informaţii, se poate consulta capitolul destinat internaţionalizării aplicaţiilor.

Toate implementările platformelor Java suportă următoarele codificări ale caracterelor:

- US-ASCII (cod ASCII american);
- ISO-8859-1 (alfabetul latin ISO, nr. 1);
- UTF-8 (format de transformare Unicode pe opt biţi);
- UTF-16BE, UTF-16LE, UTF-16 (diferite formate de transformare Unicode pe 16 biţi).

2.3.2. Traduceri lexicale

Un flux de caractere Unicode de pe o linie este tradus într-o secvenţă de atomi lexicali Java, folosind următorii trei paşi de traducere:

1. traducerea secvenţelor Unicode din fluxul caracterelor Unicode în caracterul Unicode corespunzător;
2. traducerea fluxului Unicode de la pasul 1 într-un flux de caractere de intrare şi terminatori de linie;
3. traducerea fluxului caracterelor de intrare şi a terminatorilor de linie de la pasul 2 în secvenţe de elemente de intrare Java care comprimă atomii lexicali, ce devin simboli terminali pentru gramatică sintactică Java (spaţiile şi comentariile sunt eliminate).

Pentru rezolvarea ambiguităţilor elementelor de intrare Java (identificarea corectă a atomilor lexicali), analiza lexicală se bazează pe asocierea celui mai lung cuvânt al traducerii.

Exemplul 2.3.2. Caracterele de intrare din textul `a==b` vor fi împărţite în `a`, `==`, `b`, şi nu în `a`, `=`, `=`, `b`.

Implementările Java recunosc întâi secvenţele escape de la intrare, traducând caracterele ASCII care încep cu `'\u'` şi sunt urmate de patru cifre hexazecimale, restul caracterelor fiind copiate fără schimbări.

Exemplul 2.3.3. Secvenţa `\u0064\u006f {` se traduce în secvenţa `do {`.

Apoi, implementările Java se divid în secvenţe de caractere de intrare Unicode în linii folosind terminatori de linie. Astfel se pot determina numărul liniilor folosite într-un program Java, precum şi specificarea sfârşitului comentariilor liniare (de forma `//`). Liniile sunt terminate de caracterele ASCII CR (engl. *carriage return*), LF (engl. *line feed*) sau ambele (adică CR LF). Cele două caractere, CR LF, se numără ca fiind un singur caracter. Caracterele escape asociate cu CR şi LF sunt `'\r'` şi respectiv `'\n'`.

Java defineşte trei tipuri de comentarii:

- ca şi în limbajele C/C++, textul ASCII dintre caracterele `/*` şi `*/` se numeşte *comentariu tradiţional*;
- ca şi în limbajul C++, textul ASCII de după caracterele `//` până la sfârşitul liniei se numeşte *comentariu liniar*;
- textul ASCII dintre caracterele `/**` şi `*/` se numeşte *comentariu de documentare.*

Un *identificator* este o secvenţă de lungime arbitrară (nelimitată) de litere şi cifre Java în care primul este o literă, simbolul `$` sau `_`. Un identificator nu poate fi cuvânt rezervat, literal `boolean` sau literal `null`.

Exemplul 2.3.4. `oVariabila`, `numeFunctie`, `ch`, `MititeiGustosi`, `$it` şi `_unu` sunt identificatori, pe când `2_doi` şi `10NegriMititei` nu sunt identificatori.

Un *literal* este reprezentarea codului-sursă al unei valori de tip primitiv, de tip `String` sau de tip `null`.

Un *literal întreg* poate fi exprimat în zecimal (baza 10), hexazecimal (baza 16) sau octal (baza 8). Prefixând scrierea numărului cu `0` înseamnă că numărul este scris în octal, iar cu `0x` (sau `0X`) înseamnă că numărul este scris în hexazecimal.

Un *literal în virgulă flotantă* este compus din: parte întreagă, punct zecimal, parte fracţionară, un exponent şi un sufix de tip. Exponentul este indicat de prezenţa literei `e` sau `E`. Un literal în virgulă flotantă este de tip `float`, dacă este sufixat cu litera `f` sau `F`, altfel, tipul său este `double` şi poate fi opţional sufixat de litera `d` sau `D`.

Un *literal boolean* poate fi reprezentat prin `true` şi `false` şi este întotdeauna de tip `boolean`.

Un *literal caracter* poate fi exprimat printr-un caracter sau secvenţă escape încadrate între apostrofuri şi este întotdeauna de tip `char`.

Un *literal String* constă din zero sau mai multe caractere încadrate de ghilimele, fiecare caracter putând fi reprezentat de o secvenţă escape. Un literal String este întotdeauna de tip `String` şi se referă la aceeaşi instanţă a clasei `String`.

Literalul null este de tip `null`.

Exemplul 2.3.5. Literalul întreg `28` din baza 10 este `034` (văzut astfel în baza 8) sau `0x1c` (văzut astfel în baza 16). Numărul `10000L` este un literal `long` (pe 8 octeţi). Exemple de literali în virgulă flotantă sunt `3.573` (folosind punctul zecimal), `3.67e3` (folosind notaţia ştiinţifică, numărul are valoarea `3.67 * 10^3 = 3670`), `3.567f` (indicând un literal `float`) şi `23.75d` (indicând un literal `double`). Singurii literali booleeni sunt `false` şi `true`. Exemple de literali `char` sunt `'g'`, `';'`, sau exprimând cifre hexazecimale Unicode, `'\ubab0'`, `'\u3456'`, sau prin secvenţe escape, `'\n'` (pentru linie nouă), `'\r'` (pentru Return), `'\t'` (pentru Tab), `'\b'` (pentru Backspace), `'\f'` (pentru Formfeed), `'\''` (pentru apostrof), `'\"'` (pentru ghilimele), `'\\'` (pentru Backslash).

Exemple de literali `String` sunt `"Acesta este un sir."` sau `"\u0064\u006f {"` (care se traduce în secvenţa `"do {"`).

Recunoaşterea liniei de la intrare rezultată din procesarea escape a caracterelor de intrare şi a terminatorilor de linie continuă cu reducerea la o secvenţă de elemente de intrare. Literele şi cifrele recunoscute de Java pot fi obţinute folosind metodele statice ale clasei `Character`:

```
public static boolean isLetter(char ch);
public static boolean isDigit(char ch);
```

Se poate verifica dacă este permis ca un caracter să apară la începutul, respectiv în interiorul unui identificator Java, folosind metodele:

```
public static boolean isJavaIdentifierStart(char ch);
public static boolean isJavaIdentifierPart(char ch);
```

Cuvintele rezervate Java sunt: `abstract`, `boolean`, `break`, `byte`, `case`, `catch`, `char`, `class`, `const`, `continue`, `default`, `do`, `double`, `else`, `enum`, `extends`, `final`, `finally`, `float`, `for`, `goto`, `if`, `implements`, `import`, `instanceof`, `int`, `interface`, `long`, `native`, `new`, `package`, `private`, `protected`, `public`, `return`, `short`, `static`, `super`, `switch`, `synchronized`, `this`, `throw`, `throws`, `transient`, `try`, `void`, `volatile`, `while`. Cuvintele (din limbajul C++) `const` şi `goto` sunt rezervate în Java, chiar dacă nu sunt folosite în mod curent. Literalii booleeni `true` şi `false`, respectiv literalul `null` nu sunt cuvinte rezervate.

Separatorii sunt (,), {, }, [,], ;, ,, ., iar operatorii sunt =, >, <, !, ~, ? :, :, ==, <=, >=, !=, &&, ||, ++, --, +, -, *, /, &, |, ^, %, <<, >>, >>>, +=, -=, *=, /=, &=, |=, ^=, %=, <<=, >>=, >>>=.

2.4. Tipuri de date

Java este un limbaj *puternic tipizat*, adică fiecare variabilă şi expresie are un tip cunoscut în momentul compilării. Tipurile de date limitează valorile pe care o variabilă le poate memora sau pe care le poate produce o expresie, limitează operaţiile suportate de aceste valori şi determină înţelesul acestor operaţii.

În Java, există trei categorii de tipuri de date: *primitive*, *referinţă* şi *null*. Tipurile primitive sunt tipurile *numerice* şi `boolean`. Tipurile numerice sunt tipurile *integrale* (`byte`, `short`, `int`, `long` şi `char`) şi tipurile în *virgulă flotantă* (`float` şi `double`). Tipurile referinţă sunt tipurile *clasă*, *interfaţă* şi *tablou*. Un obiect în Java este o instanţă creată dinamic a unui tip clasă sau a unui tablou creat dinamic. Valorile unui tip referinţă sunt referinţe la obiecte. Toate obiectele, inclusiv tablourile, suportă metode ale clasei `Object`. Literalii `String` sunt reprezentaţi de obiecte `String`.

Tipul `null` este special, acesta neavând nume. Este deci imposibil să declarăm o variabilă de tip `null` sau să o convertim la o expresie de tip `null`. Referinţa `null` este o valoare posibilă pentru o expresie de tip `null` şi poate fi convertită la orice tip referinţă.

Numele tipurilor sunt folosite în declaraţii, conversii, în expresii de creare a instanţei unei clase, în expresii de creare a tablourilor şi în expresii folosite de operatorul `instanceof`.

Valorile primitive nu împart starea cu alte valori primitive, deci o variabilă de tip primitiv va ţine o valoare primitivă de acel tip. Valoarea unei variabile de tip primitiv poate fi schimbată doar prin operaţii de atribuire a acelei variabile.

Referitor la tipurile primitive de date, avem următoarele informaţii:

Tip	Dimensiunea reprezentării (număr de biţi)	Valoarea minimă a domeniului	Valoarea maximă a domeniului
`boolean`	1	`false`	`true`
`byte`	8	$-2^7 = -128$	$2^7-1 = 127$
`short`	16	$-2^{15} = -32768$	$2^{15}-1 = 32767$
`int`	32	$-2^{31} = -2147483648$	$2^{31}-1 = 2147483647$
`long`	64	$-2^{63} = -9223372036854775808$	$2^{63}-1 = 9223372036854775807$
`char`	16	\u000 = 0	\uffff = 65535
`float`	32	[–0x7f7fffff, –0x1]	[0x1, 0x7f7fffff]
`double`	64	[–0x7fefffffffffffff, –0x1]	[0x1, 0x7fefffffffffffff]

Orice valoare a oricărui tip integral poate fi convertită la sau către un tip numeric. În schimb, nu există conversii între tipurile integrale şi tipul `boolean`. Depăşirea domeniului implică atribuirea unei valori care nu concordă cu realitatea matematică.

Exemplul 2.4.1. Fie secvenţa de cod Java:

```
int a = 1400000000, b = 1600000000;
System.out.println(a + b);
```

Aceasta va afişa: `-1294967296` care corespunde cu interpretarea domeniului unui tip de date cu un buffer circular. Astfel, valoarea reală de `3000000000` depăşeşte domeniul {-2147483648, ..., 0, ..., 2147483647}. Cum valoarea `2147483648` corespunde cu `-2147483648`, rezultă că `3000000000` va fi interpretată ca fiind: `-2147483648 + (3000000000 - 2147483648) = -1294967296.`

Există şi depăşiri de domeniu care sunt detectate încă de la compilare.

Exemplul 2.4.2. La compilarea instrucţiunii:

```
short u = 32768;
```

se obţine eroarea la compilare: „`possible loss of precision`”. O rezolvare a acestei erori este să realizăm o conversie explicită a literalului `32768` la `short`. Aşa cum este de aşteptat, se va tipări `-32768` care este o valoare eronată (ca în exemplul 2.4.1).

```
short u = (short) 32768;
System.out.println("u = " + u);
```

Tipurile `float` şi `double` respectă standardul ANSI/IEEE (engl. *American National Standard Institute/Institute of Electrical and Electronics Engineers*) numărul 754/1985, referitor la valorile şi operaţiile în virgulă mobilă. Astfel, tipul `float` are domeniul de valori între [- 3.4E + 38, - 1.4E - 45] U [1.4E - 45, 3.4E + 38] şi furnizează o precizie de 6-7 cifre semnificative, pe când tipul `double` are domeniul de valori între [- 1.79E + + 308, - 4.94E - 324] U [4.94E - 324, 1.79E + 308] şi furnizează o precizie de 14-15 cifre semnificative.

Clasele `Float` şi `Double` pun la dispoziţie constante şi metode membre importante pentru domeniul de valori, astfel:

- `MAX_VALUE`, `MIN_VALUE` (pentru precizarea valorilor extreme pozitive);
- `POSITIVE_INFINITY`, `NEGATIVE_INFINITY` (pentru precizarea valorilor infinite);
- `NaN` (engl. *Not-a-Number*), folosită pentru rezultatul operaţiilor speciale, cum ar fi împărţirea la zero.

Valorile negative ale domeniului sunt `MAX_VALUE`, respectiv `MIN_VALUE`. La numere zecimale, împărţirea la zero se poate preveni folosind metoda (din clasele `Double` sau `Float`):

```
public static boolean isInfinite(Double x)
```

care returnează `true`, dacă argumentul `x` este infinit.

Exemplul 2.4.3. Următoarea secvenţă de cod testează împărţirea la zero:

```
double x = 2, y = 0;
if (Double.isInfinite(x / y))
  System.out.println("infinit");
else
  System.out.println(x / y);
```

Obţinerea numărului `NaN` poate fi testată folosind metoda `isNaN()` din clasele `Float` sau `Double` (cu sau fără argument):

```
public static boolean isNaN(double v);
```

Situaţiile particulare care pot apărea la împărţiri între două numere zecimale sunt exprimate în tabel:

x	y	x / y	x % y
finit	± 0.0	± infinit	NaN
finit	± infinit	± 0.0	valoare lui x
± 0.0	± 0.0	NaN	NaN
± infinit	finit	± infinit	NaN
± infinit	± infinit	NaN	NaN

Exemplul 2.4.4. Folosind metoda `isNaN()`, putem preveni operaţiile fără sens:

```
double x = 0, y = 0;
if (Double.isNaN(x / y))
  System.out.println("operatie care conduce la obtinerea "+
     "unui nenumar!");
```

Exemplul 2.4.5. Putem testa dacă valoarea unui număr zecimal este prea mică astfel:

```
if (Math.abs(a) < 0.1E20f)  . . .
```

Exemplul 2.4.6. Putem testa dacă aplicarea funcţiei radical de ordin 2 este corectă (argumentul este pozitiv):

```
float a = 0, b = 1, d = -5;
double radical = Math.sqrt(d);
if (! Double.isNaN(radical)) {
  double radacina = ( -b + radical) / (2.0f * a);
  System.out.println(radacina);
}
else
  System.out.println("d este negativ sau este NaN");
```

O operaţie în virgulă flotantă care depăşeşte domeniul produce infinit cu semn sau zero pozitiv. De asemenea, operaţiile aritmetice care nu au rezultat definit matematic returnează `NaN`. Operaţiile numerice care au un operand `NaN` returnează `NaN`. Deoarece `NaN` nu induce o ordine, operaţiile de comparaţie care conţin `NaN` returnează `false`, cu excepţia lui `!=` care returnează `true`.

Rezultatul operaţiei în virgulă mobilă este rotunjit la numărul cel mai apropiat conform cu precizia tipului de date respectiv.

Exemplul 2.4.7. Următorul program Java pune în evidenţă depăşirea domeniului valorilor în virgulă flotantă, precum şi rotunjirea făcută în unele situaţii:

```
public class NumereInVirgulaMobila {
  public static void main(String args[]) {
    // depasire a domeniului catre plus infinit
    double infinitMare = 1e308;
    System.out.println(infinitMare * 10);
    // depasire a domeniului catre zero pozitiv
    double infinitMic = 1e-323;
    System.out.println(infinitMic / 10);
    // rotunjire a unui rezultat inexact de tip float
    // matematic, ar fi trebuit sa se afiseze 1.0
    float rezultatInexactFloat = 1.0f / 41;
    System.out.println(rezultatInexactFloat * 41);
```

```
        // rotunjire a unui rezultat inexact de tip double
        // matematic, ar fi trebuit sa se afiseze 1.0
        double rezultatInexactDouble = 1.0f / 49;
        System.out.println(rezultatInexactDouble * 49);
    }
}
```

Rezultatul afişat de execuţia acestui program este:

```
Infinity
0.0
0.99999994
0.9999999795109034
```

Continuăm cu prezentarea tipului referinţă. Un obiect este o instanţă a unei clase sau un tablou. Valorile referinţă sunt pointeri la aceste obiecte sau sunt egale cu referinţa specială `null` (care nu se referă la nici un obiect). O instanţă a unei clase este explicit creată folosind o expresie de creare a unei instanţe sau prin apelul metodei `newInstance()` din clasa `Class`. Un tablou este explicit creat printr-o expresie de creare a unui tablou. Un obiect de tip tablou este creat implicit când se evaluează o expresie de iniţializare a unui tablou. Aceasta poate apărea când este iniţializată o clasă sau o interfaţă, când este creată o nouă instanţă a unei clase sau când se execută o instrucţiune de declarare a unei variabile locale. Declararea şi definirea claselor şi tablourilor vor fi prezentate în detaliu în capitolele următoare.

Exemplul 2.4.8. Următorul exemplu pune în evidenţă crearea unor instanţe pentru o clasă:

```
class Exemplu {
  int x;

  // constructor fara parametri
  Exemplu() {
    System.out.println("constructor fara parametri");
  }

  // alt constructor
  Exemplu(int x) {
    this.x = x;
  }

  // instanta creata explicit la momentul initializarii clasei
  static Exemplu primulX = new Exemplu(1);

  // metoda de actualizare a campului x pentru obiectul curent
  public void setX(int x) {
    this.x = x;
    }

    // metoda necesara pentru afisarea explicita a unei
    // instante din clasa Exemplu
    public String toString() {
      return "x = " + x ;
    }
  } // sfarsitul definitiei clasei Exemplu

  public class DiferiteInstante {
    public static void main(String[] args) {
      System.out.println("primulX: " + Exemplu.primulX);

      // declarare variabile
      Exemplu obiectDoi = null;

      // creare explicita folosind newInstance()
      try {
        obiectDoi = (exemplu)
          Class.forName("Exemplu").newInstance();
        System.out.println("obiectDoi: " + obiectDoi);
        obiectDoi.setX(2);
        System.err.println("obiectDoi: " + obiectDoi);
      }
      catch (Exception e) {
        System.out.println(e);
      }

      // crearea unui obiect folosind operatorul new
      Exemplu obiectTrei = new Exemplu(3);
      System.out.println("obiectTrei: " + obiectTrei);
    }
  } // sfarsitul definitiei clasei DiferiteInstante
```

Execuţia acestui program Java conduce la ieşirea:

```
primulX: x = 1
constructor fara parametri
obiectDoi: x = 0
obiectDoi: x = 2
obiectTrei: x = 3
```

Clasa `Object` este o superclasă pentru toate clasele Java. O variabilă de tip `Object` poate ţine o referinţă a oricărui obiect, fie că este o instanţă a unei clase sau a unui tablou. Toate tipurile clasă şi tablou moştenesc metodele clasei `Object`.

Instanţele clasei `String` reprezintă secvenţe de caractere Unicode. Un obiect `String` are o valoare constantă (care nu se schimbă în timpul execuţiei programului).

Literalii `String` sunt referinţe la instanţe ale clasei `String`. Operatorul de concatenare a şirurilor, +, creează implicit un nou obiect `String`.

Două tipuri referinţă sunt de acelaşi tip dacă sunt tipuri tablou ale căror componente sunt de acelaşi tip sau sunt tipuri clasă sau interfaţă, sunt încărcate de acelaşi încărcător de clasă şi au acelaşi nume (se mai spune că sunt în aceeaşi clasă sau interfaţă).

Exemplul 2.4.9. Exemplul de mai jos pune în evidenţă utilizarea tipurilor de date Java:

```
import java.io.IOException;
import java.util.Random;

public class Zaruri {

  public static void main(String[] arg) {
    // instantieri
    Random r = new Random();
    int[] zar = new int[2];
    for (int k = 0; k < 10; k++) {
     for (int i = 0; i < 2; i++) {
         zar[i] = (int)(r.nextDouble() * 6) + 1;
     }
     System.out.println("\nZarurile sunt: " + zar[0] + " " +
         zar[1]);
     if (zar[0] == zar[1])
         System.out.println(
           "Ati castigat! Zarurile sunt egale!");
     else
         System.out.println(
           "Ati pierdut! Zarurile nu sunt egale!");

     // asteptam apasarea tastei ENTER
     System.out.print(
          "Apasati tasta ENTER pentru a continua...");
     try {
         // citim un caracter de la tastatura
         System.in.read();
         // eliminam celelalte caractere citite
     System.in.skip(System.in.available());
      }
     catch(IOException e) {
         System.err.println(e);
     }
    }
```

```
        System.out.println("Aplicatia s-a terminat!");
    }
} // sfarsitul definitiei clasei Zaruri
```

Tipurile de date sunt folosite în următoarele declaraţii:

1. Tipuri importate: de exemplu, tipul `Random` (s-a importat clasa `java.util.Random`);
2. Tipuri folosite pentru parametrii metodelor: de exemplu, tipul `String []` este utilizat de metoda `main()`;
3. Tipuri folosite pentru variabile locale: tipul primitiv `int` este folosit pentru variabila locală `i` a metodei `main()`;
4. Tipuri folosite pentru crearea unei instanţe a unei clase: tipul `Random` la iniţializarea variabilei `r`;
5. Tipuri folosite pentru crearea tablourilor: tipul `int` este folosit pentru crearea tabloului `zar` cu două elemente;
6. Tipuri folosite pentru conversii explicite: conversia la tipul `int` este utilizată în cea de-a doua instrucţiune `for`, pentru a ne asigura că valorile zarurilor sunt numere întregi între 1 şi 6.

2.5. Expresii şi operatori

Un program Java se bazează în principal pe *evaluarea expresiilor*, fie prin *efectele laterale* (asignarea de valori variabilelor în timpul evaluării expresiilor), fie pentru *valorile returnate* de acestea (utilizate ca argumente sau operanzi în alte expresii) sau pentru *afectarea secvenţei de execuţie* în instrucţiuni.

Când se evaluează o expresie într-un program Java, rezultatul întors poate fi o variabilă, o valoare, sau nimic (engl. *void*). Deoarece expresiile pot conţine asignări interne, operatori de incrementare/decrementare şi apeluri de metode, rezultă că evaluarea unei expresii poate produce efecte laterale. O expresie poate să întoarcă `void` în situaţia apelurilor de metode care întorc `void`, caz în care expresia este instrucţiune şi nu serveşte la transmiterea valorii sale. Fiecare expresie apare într-o declaraţie a unui tip clasă sau interfaţă care poate fi iniţializator de câmp, iniţializator static, declaraţie a unui constructor sau în codul unei metode.

Dacă o expresie este o variabilă sau o valoare, atunci expresia trebuie să aibă tipul cunoscut încă de la compilare. Ca şi la variabile, valoarea unei expresii este compatibilă cu tipul expresiei. Astfel, valoarea unei expresii al cărei tip este T poate fi asignată unei variabile de tip T.

Dacă tipul unei expresii este un tip primitiv, atunci valoarea expresiei este de acelaşi tip primitiv. Dacă tipul expresiei este un tip referinţă, atunci clasa şi valoarea obiectului referenţiat nu se cunosc la momentul compilării, ci al execuţiei.

Pentru a da definiţia formală a *expresiei*, mai întâi definim noţiunea de *expresie primară*. Expresiile primare sunt literalii, accesul la câmpuri şi tablouri, apelurile de metode, expresiile parantetizate. Un *operator* cu *n* argumente este o funcţie cu *n* argumente care se mai numesc şi *operanzi*. Utilizând o manieră recursivă de descriere, o *expresie compusă* (sau, simplu, *expresie*) poate fi expresie primară, o expresie care foloseşte operatori unari, binari şi ternari sau o expresie de conversie. De exemplu, operatorul + este unar şi binar, operatorul condiţional `?:` este ternar etc.

În continuare, vom prezenta *instrucţiunea de atribuire*. O instrucţiune de asignare are sintaxa:

```
<numeVariabila> = <expresie> ;
```

unde = se numeşte *operator de atribuire*. Semantica acestei instrucţiuni este simplă: se evaluează `<expresie>` şi se memorează valoarea acesteia la adresa variabilei `<numeVariabila>`.

Exemplul 2.5.1. În urma execuţiei instrucţiunilor de atribuire:

```
int a = 4, b = 7;
int c = a + b;
```

se obţine valoarea `11` pentru variabila `c`. Acestea se pot scrie într-o „singură instrucţiune" astfel:

```
int a, b, c = (a = 4) + (b = 7);
```

Expresiile `(a = 4)` şi `(b = 7)` întorc valoarea părţii drepte, adică 4 şi, respectiv, `7`. Astfel, se face asignarea tuturor celor trei variabile într-o manieră compactată.

O altă versiune a instrucţiunii de asignare este:

```
<numeVariabila> <operator>= <expresie> ;
```

unde `<operator>` poate fi: +, -, *, /, `%`, <<, >>, >>>, `&`, | şi ^, între `<operator>` şi = nefiind spaţii (în caz contrar, se obţine eroare la compilare). Aceasta este o prescurtare a instrucţiunii:

```
<numeVariabila> = <numeVariabila> <operator> (<expresie>);
```

Secvenţa execuţiei operatorilor cu aceeaşi precedenţă dintr-o instrucţiune este determinată de *proprietatea de asociativitate*. Astfel, în Java există două tipuri de asociativităţi: *stângă* şi *dreaptă*. Spunem că <op> este operator stâng-asociativ dacă pentru orice x, y, z avem `x <op> y <op> z = (x <op> y) <op> z`. Spunem că <op> este operator drept-asociativ dacă pentru orice x, y, z avem `x <op> y <op> z = x <op> (y <op> z)`.

Se mai spune că operatorii stâng-asociativi se evaluează de la stânga la dreapta, iar cei drept-asociativi de la dreapta la stânga. De exemplu, operatorul =, precum şi cei de forma `<operator>=` sunt drept-asociativi.

Exemplul 2.5.2. Fie instrucţiunile de asignare:

```
int a = 1, b, c = 3, d;
a += b = c += d = 5;
```

Cea de-a doua linie se poate rescrie astfel:

```
a += (b = (c += (d = 5)));
```

Astfel, se obţin prin efecte laterale valorile `9`, `8`, `8`, `5` pentru variabilele `a`, `b`, `c` şi `d`. În Java, operanzii se evaluează într-o ordine predefinită, şi anume de la stânga la dreapta. De obicei, fiecare expresie are cel mult un efect lateral şi codul nu este dependent de excepţia care apare drept consecinţă a evaluării expresiilor de la stânga la dreapta. În cazul operatorilor binari, operandul din stânga este complet evaluat înaintea evaluării oricărei părţi a operandului din dreapta.

Exemplul 2.5.3. Execuţia următorului program Java:

```
public class EvaluareStanga {
  public static void main(String[] args) {
    int a = 5;
    int b = (a = 4) * a;
    System.out.println(b);
  }
}
```

va afişa `16`, deoarece se evaluează mai întâi subexpresia `(a = 4)` care întoarce `4`. Mai mult, valoarea lui `a` este `4`, şi nu `5` cum era iniţial. Astfel, valoarea lui `b` este `4 * 4`, adică `16`.

Continuăm cu prezentarea operatorilor, urmând ca în secţiunea 2.6 să prezentăm noţiunea de variabilă. Vom prezenta operatorii după tipurile operanzilor pe care îi pot avea.

Începem cu operatorii care lucrează cu operanzi de tipuri integrale:

1. *Operatorii de comparaţie*, care întorc o valoare de tip `boolean`: <, <=, >, >=, ==, !=
2. *Operatorii numerici*, care întorc o valoare de tip `int` sau `long`: +, - (unari şi binari), *, /, `%` (multiplicativi), ++, -- (incrementare şi decrementare, formele prefixată şi postfixată), <<, >>, >>> (deplasare cu şi fără semn), ~ (complement faţă de 2), `&`, `|`, ^ (şi, sau inclusiv, sau exclusiv pe biţi)
3. *Operatorul condiţional* `? :`
4. *Operatorul de conversie explicită*, care poate converti o valoare integrală la o valoare de un tip numeric specificat.
5. *Operatorul de concatenare a şirurilor*, care, în cazul unui operand de tip `String` şi al unuia de tip integral, va face conversia celui integral la `String`, creând în final un nou obiect `String` egal cu concatenarea acestora.

Dacă un operator întreg (diferit de cel de deplasare) are un operand de tip `long`, atunci operaţia este efectuată folosind o precizie de 64 de biţi şi convertind celălalt operand la tipul `long`, valoarea returnată de operatorul numeric fiind de tip `long`. În caz contrar, operaţia este efectuată folosind o precizie de 32 de biţi, convertind operanzii la tipul `int`, rezultatul operatorului numeric fiind de tip `int`. Operatorii întregi predefiniţi nu indică depăşirea domeniului (exemplul 2.4.1), singurele excepţii de tip `ArithmeticException` care pot apărea se pot datora împărţirii la zero în cazul operatorilor / şi `%`.

Exemplul 2.5.4. Următorul program va scoate în evidenţă apariţia unei excepţii aritmetice:

```
public class ExceptiiNumerice {
  public static void main(String args[]) {
    int a = 3, b = 0;
    try {
      System.out.println("a / b = " + a / b);
    }
    catch (ArithmeticException e) {
     System.err.println("Nu este permisa impartirea la zero ("
        + e.getMessage() + ")");
    }
  }
} // sfarsitul definitiei clasei ExceptiiNumerice
```

Operatorii de comparaţie sunt stâng-asociativi, însă în marea majoritate a cazurilor necesită paranteze.

Exemplul 2.5.5. Expresia `1 < 2 < 3` se interpretează `(1 < 2) < 3`, care se evaluează la `true < 3` şi conduce la o eroare de compilare, deoarece operatorul < nu lucrează cu operanzi de tip `boolean`.

În cazul operatorilor de comparaţie, tipul fiecărui operand trebuie să fie tip numeric primitiv, altfel se obţine o eroare de compilare. Evaluarea expresiilor care conţin operatori de comparaţie se realizează în mod natural. De exemplu, pentru operatorul < avem `<operandUnu> < <operandDoi>` se evaluează la `true`, dacă valoarea lui `<operandUnu>` este mai mică decât valoarea lui `<operandDoi>`.

Pentru ceilalţi operatori, lucrurile decurg asemănător (<= este „mai mic sau egal", > este „mai mare", >= este „mai mare sau egal", == este „egal cu" şi `!=` este „diferit de"). Operatorii == şi `!=` se mai numesc şi *operatori de egalitate* şi se deosebesc de ceilalţi operatori prin faptul că au prioritatea mai mică.

Exemplu 2.5.6. Datorită priorităţii mici a operatorului ==, expresia `1 < 2 == 3 < 4` se parsează ca fiind `(1 < 2) == (3 < 4)`. Aceasta se evaluează la `true == true`, care returnează `true`. (A parsa = a procesa, a analiza etc.)

Tipul unei expresii de egalitate este întotdeauna `boolean`. În toate cazurile, `a != b` este echivalent semantic cu `!(a == b)`. Operatorii de egalitate sunt comutativi, dacă expresiile operanzilor nu au efecte laterale.

Referitor la operatorul unar de adunare +, putem spune că tipul expresiei operandului + trebuie să fie un tip numeric primitiv, altfel obţinem o eroare la compilare. Tipul expresiei + unar va fi tipul convertit la compilare, iar rezultatul expresiei + unar nu este o variabilă, ci o valoare (chiar dacă rezultatul expresiei operand este o variabilă).

În ceea ce priveşte operatorul unar de scădere, acesta este asemănător cu cel unar de adunare. Deoarece Java utilizează reprezentarea în complement faţă de doi pentru întregi şi domeniul valorilor complement faţă de doi nu este simetric, rezultă că negarea valorii maxime negative pentru tipurile `int` şi `long` iese din domeniu, afişând valoarea maximă negativă (fără a arunca vreo excepţie). Pentru toate valorile întregi, `x` este egal cu `(~x)+1`.

Exemplul 2.5.7. Execuţia următorului subprogram Java:

```
System.out.println(Integer.MAX_VALUE + " " + -Integer.MAX_VALUE);
System.out.println(Long.MAX_VALUE + " " + -Long.MAX_VALUE);
```

va afişa:

```
2147483647 -2147483647
9223372036854775807 -9223372036854775807
```

Operatorii +, – se mai numesc *operatori de adunare* şi se aplică operanzilor numerici de acelaşi tip, producând suma, respectiv diferenţa acestora. Adunarea este o operaţie comutativă (pentru orice `a`, `b` avem `a + b = b + a`), dacă expresiile operanzilor nu au efecte laterale. Adunarea întregilor este asociativă când toţi operanzii sunt de acelaşi tip (nu este cazul adunării numerelor în virgulă flotantă). Dacă o adunare întreagă depăşeşte domeniul de valori, atunci rezultatul va corespunde unei sume matematice reprezentate în formatul complement faţă de doi şi, eventual, va avea alt semn (exemplul 2.4.1). Operatorul binar de scădere, –, foloseşte în definiţia sa operatorii binari de adunare şi unar de scădere astfel: pentru orice `a`, `b` avem `a - b = a + (-b)`.

Operatorii `*`, `/` şi `%` se numesc *operatori multiplicativi* şi se aplică operanzilor numerici de acelaşi tip, executând produsul (înmulţirea), raportul (împărţirea), respectiv restul împărţirii acestora. Aceşti operatori au aceeaşi precedenţă şi sunt stâng-asociativi. Tipul fiecărui operand al unui operator multiplicativ trebuie să aparţină unui tip numeric primitiv, altfel obţinem o eroare de compilare. Înmulţirea este o operaţie comutativă, dacă expresiile operanzilor nu au efecte laterale. Înmulţirea întregilor este asociativă când toţi operanzii sunt de acelaşi tip (nu este cazul înmulţirii numerelor în virgulă flotantă). Dacă o înmulţire întreagă depăşeşte domeniul de valori, atunci rezultatul va corespunde unui produs matematic reprezentat în formatul complement faţă de doi şi, eventual, va avea alt semn (exemplul 2.4.1).

Pentru operanzi întregi, operatorul `/` întoarce partea întreagă a împărţirii primului operand la cel de-al doilea. Astfel, notând `a / b = c` (`c` fiind câtul dintre `a` şi `b`) şi `a % b = d` (`d` fiind restul împărţirii lui `a` la `b`), putem spune că `c` are semnul produsului dintre `a` şi `b`, iar `d` are semnul lui `a`. Acest lucru previne utilizarea greşită a teoremei împărţirii cu rest din matematică (`a = b * c + d`, unde `0 <= d < b`). Dacă valoarea lui `b` este 0, atunci se aruncă o excepţie `ArithmeticException`.

Exemplul 2.5.8. Avem: `7 / 2 = 3`, `7 % 2 = 1`, `(-7) / 2 = -3`, `(-7) % 2 = -1`, `7 / (-2) = -3`, `7 % (-2) = 1`, `(-7) / (-2) = 3`, `(-7) % (-2) = -1`.

Referitor la operatorii (unari) de incrementare/decrementare (++, --), aceştia pot apărea în formă postfixată (operatorul este la sfârşit) sau prefixată (operatorul este la început). Rezultatul expresiei unare trebuie să fie o variabilă a unui tip numeric, altfel apare o eroare de compilare. Tipul expresiei de incrementare/decrementare este identic cu tipul variabilei. În cazul operatorilor postfixaţi, de exemplu a++ sau a--, se va aduna/scădea valoarea 1 la vechea valoare a variabilei a, aceasta modificându-şi valoarea cu 1. Valoarea expresiei a++ sau a-- este vechea valoare a variabilei a (adică înaintea modificării ei). Ca şi la forma postfixată, în cazul operatorilor prefixaţi, de exemplu ++a sau --a, se va aduna/scădea valoarea 1 la vechea valoare a variabilei a, aceasta modificându-şi valoarea cu 1. Valoarea expresiei ++a sau --a este noua valoare a variabilei a (adică modificarea ei).

Rezultatul expresiei de incrementare/decrementare este o valoare şi nu o variabilă. În plus, este evident că operanzii operatorilor ++/-- nu pot fi constanţi (declaraţi `final` sau chiar literali). Astfel, nu putem scrie `int i = 4++;`.

Exemplul 2.5.9. Iată câteva exemple de utilizare a operatorilor unari de incrementare/decrementare:

```
int i = 5;
System.out.println(++i + " " + ++i + " " + ++i);
```

Datorită evaluării de la stânga la dreapta a expresiei argument din metoda `println()`, rezultă că se va afişa:

```
6 7 8
```

Continuăm cu liniile de cod Java:

```
int i = 3, j = 7, k = ++i + j--;
System.out.println(k);
```

În urma execuţiei acestor ultime două linii se va afişa 11.

Continuăm cu prezentarea operatorilor de deplasare. Astfel, << este operatorul de deplasare stânga, >> este operatorul de deplasare dreapta cu semn, iar >>> este operatorul de deplasare dreapta fără semn. Aceştia sunt stâng-asociativi, operandul din dreapta specificând numărul de poziţii cu care se deplasează. Tipul fiecărui operand al unui operator de deplasare trebuie să fie de tip integral, altfel apare o eroare de compilare. Operandul din dreapta este redus modulo 32, dacă operandul din stânga este (sau poate fi convertit la) `int`, sau modulo 64, dacă operandul din stânga este (sau poate fi convertit la) `long`. Valoarea expresiei `a << b`, unde b este întreg pozitiv, este egală cu valoarea lui a deplasată cu b poziţii spre stânga, care este echivalentă cu a înmulţit cu 2^b. Valoarea expresiei `a >> b` este egală cu valoarea lui a deplasată cu b poziţii spre dreapta, care pentru numere pozitive este echivalentă cu partea întreagă a lui a împărţit la 2^b. Valoarea lui `a >>> b` este egală cu valoarea lui a deplasată la dreapta cu b poziţii. Dacă a este pozitiv, atunci rezultatul coincide cu `a >> b`. Dacă a este negativ, rezultatul lui `a >>> b` este egal cu expresia `(a >> b) + (2 << ~s)`, dacă tipul operandului stâng este `int`, sau `(a >> b) + (2L << ~s)`, dacă tipul operandului stâng este `long`. Termenul care se adună (este vorba de `(2 << ~s)`, respectiv `(2L << ~s)`) are rolul de a inhiba propagarea din stânga a simbolurilor `1`.

Exemplul 2.5.10. Dacă avem:

```
int a = -5;
int b = a >>> 1;
```

atunci b nu este un număr negativ, cum se întâmplă în cazul operatorului de deplasare dreapta >>, şi va avea valoarea `2147483645`.

Exemplul 2.5.11. Ne aşteptăm ca expresia `-64 >>> 4` să aibă valoarea `12`, deoarece `-64` are reprezentarea binară `11000000`, iar după deplasarea la dreapta cu 4 biţi să obţinem `00001100`. Din păcate, lucrurile nu stau deloc aşa. Mai întâi se face conversia lui `-64` (care este literal de tip `byte`) la `int`. Deci acesta va fi în memorie (în binar):

```
11111111 11111111 11111111 11000000
```

Ne deplasăm la dreapta cu 4 biţi şi obţinem:

```
00001111 11111111 11111111 11111100
```

Se observă că ultimul octet diferă la cele două reprezentări (`11000000` faţă de `11111100`). Astfel, valoarea întoarsă de expresia `-64 >>> 4` va fi de tip `int` şi va avea valoarea `268435452`. De altfel, în Java nu putem scrie instrucţiuni de genul:

```
byte b = -64 >>> 4;
```

deoarece obţinem eroare la compilare (engl. *Explicit cast to int is required!*). Explicaţia este că valoarea `-64` se memorează pe mai mult de un octet şi deci la conversia implicită s-ar pierde din precizie.

Continuăm cu prezentarea operatorului unar de complementariere. Tipul expresiei operand a operatorului `~` trebuie să fie un tip integral, altfel apare o eroare de compilare. Tipurile `short`, `byte` se convertesc implicit la `int`, astfel că expresia `~a` are tipul `int` sau `long`. Regulile de complementariere aplicate alfabetului `{0, 1}` sunt: `~0 = 1` şi `~1 = 0`. Acestea se pot generaliza la cuvinte de lungime `32`, respectiv `64`, astfel: $\sim(a_1a_2...a_n) = (\sim a_1)(\sim a_2)...(\sim a_n)$.

Exemplul 2.5.12. Este clar că aplicând operatorul de complementariere unui număr pozitiv, acesta devine negativ şi reciproc. Acest lucru se datorează faptului că şi bitul de semn devine contrar. Astfel:

```
~ 11100111 10111000 10101001 11000010 =
     = 00011000 01000111 01010110 00111101
```

Valoarea numerelor pozitive din memoria calculatorului se deduce din teorema fundamentală a aritmeticii ($(a_1a_2,..a_{n(d)} = a_1*d^{n-1} + a_2*d^{n-2} + ... + a_n)$). Pentru numere negative se foloseşte în plus regula `~x = (x) - 1`, de unde deducem că:

```
-x = ~x + 1.
```

Exemplul 2.5.13. Numerele pozitive au primul bit (se mai numeşte bit de semn) egal cu `0`. Astfel,

`a = 00011000 01000111 01010110 00111101` $= 1 + 2^2 + 2^3 + 2^4 + 2^5 + 2^9 + 2^{10} + 2^{12} + 2^{14} + 2^{16} + 2^{17} + 2^{18} + 2^{22} + 2^{27} + 2^{28} = 407328317$

Numerele negative au primul bit egal cu `1`, iar valoarea lor o vom calcula cu formula `x = -(~x + 1)`. Astfel, pentru calculul lui `b = 11011001 01001101 01011110 10110101` vom calcula valoarea complementului său, care este un număr pozitiv: `~b = 00100110 10110010 10100001 01001010` $= 2^1 + 2^3 + 2^6 + 2^8 + 2^{13} + 2^{15} + 2^{17} + 2^{20} + 2^{21} + 2^{23} + 2^{25} + 2^{26} + 2^{29} = 649240906$.

Adunăm acum o unitate şi apoi schimbăm semnul, obţinând astfel `b = -649240907`. Pentru obţinerea formei binare a unui literal integral, putem utiliza o metodă din clasa `Integer` (sau `Long`), şi anume:

```
public static String toBinaryString(int i);
```

care întoarce reprezentarea din memorie a numărului `i` (în baza doi, fără a pune în evidenţă zerourile nesemnificative). Astfel, executând programul Java de mai jos:

```
public class FormaBinara {
   public static void main(String args[]) {
      System.out.println(Integer.toBinaryString(407328317));
      System.out.println(Integer.toBinaryString(-649240907));
   }
}
```

facem verificarea calculelor matematice de mai sus:

```
11000010001110101011000111101
11011001010011010101111010110101
```

Continuăm acum cu operatorii pe biţi „şi” `&`, „sau inclusiv” `|`, „sau exclusiv” `^`. Aceşti operatori sunt stâng asociativi, dar au precedenţe diferite, operatorul `&` având cea mai mare prioritate şi operatorul `|` având cea mai mică prioritate. În plus, aceşti operatori sunt comutativi dacă expresiile operanzilor nu au efecte laterale. Când ambii operanzi sunt de tip integral, atunci aceştia se convertesc la `int` sau `long`, după caz. Aceştia au definiţiile matematice obişnuite, astfel:

1. `a & b = 1` dacă şi numai dacă `a = 1` şi `b = 1`;
2. `a | b = 1` dacă şi numai dacă `a = 1` sau `b = 1`;
3. `a ^ b = 1` dacă şi numai dacă `a = 1` sau `b = 1` şi `a` este diferit de `b`.

Definiţia 2 se poate da echivalent: `a | b = 0` dacă şi numai dacă `a = 0` şi `b = 0`. La fel, definiţia 3 se poate rescrie astfel: `a ^ b = 1` dacă şi numai dacă `a + b = 1`.

Exemplul 2.5.14. Fie instrucţiunile:

```
int a = 3333, b = 7777;
```

Iată care sunt reprezentările în memorie, respectiv valorile corespunzătoare în baza 10:

Expresie	Reprezentare binară	Valoare zecimală
`a`	`00000000 00000000 00001101 00000101`	`3333`
`b`	`00000000 00000000 00011110 01100001`	`7777`
`a&b`	`00000000 00000000 00001100 00000001`	`3073`
`a^b`	`00000000 00000000 00010011 01100100`	`4964`
`a|b`	`00000000 00000000 00011111 01100101`	`8037`
`~(a|b)`	`11111111 11111111 11100000 10011010`	`-8038`
`~a&~b`	`11111111 11111111 11100000 10011010`	`-8038`

Se observă că se verifică relaţiile lui De Morgan, cum ar fi `~(a|b)=~a&~b`. Putem considera şi operaţii pentru numere scrise în baza 16, cum ar fi `0xff00 & 0xf0f0 = 0xf000`, `0xff00 | 0xf0f0 = 0xfff0` şi `0xff00 ^ 0xf0f0 = 0x0ff0`.

Continuăm cu operatorul condiţional `?:` care apare în expresii (condiţionale) de forma: `<expresieBooleana> ? <expresie> : <expresie>`, unde `<expresie>` poate fi la rândul ei expresie condiţională. Operatorul condiţional este drept-asociativ, spre deosebire de majoritatea operatorilor. `<expresieBooleana>` trebuie să fie de tip `boolean`, altfel apare o eroare de compilare. Operatorul condiţional poate fi folosit pentru a alege dintre operanzii al doilea şi al treilea de tip numeric, de tip `boolean`, de tip referinţă sau de tip `null`. Toate celelalte cazuri conduc la o eroare de compilare. Valoarea întoarsă de o expresie condiţională este determinată la execuţie astfel: mai întâi se evaluează expresia booleană; dacă aceasta se evaluează la `true`, atunci se evaluează operandul al doilea a cărui valoare se returnează; în caz contrar (dacă aceasta se evaluează la `false`), se returnează valoarea celui de-al treilea operand. Deoarece operatorul condiţional nu evaluează toţi operanzii, se mai spune că acesta este un operator „leneş" (engl. *lazy*).

Exemplul 2.5.15. Expresia `1<2?3:4<5?6:7` se interpretează astfel: `(1<2)?3:((4<5)?6:7)`, care se evaluază la `3`.

Referitor la tipul întors de expresia condiţională, avem următoarele reguli:

1. dacă al doilea şi al treilea operand au acelaşi tip (poate fi şi tipul `null`), atunci acesta va fi tipul returnat de expresia condiţională;
2. altfel, dacă al doilea şi al treilea operand au tip numeric, atunci avem subcazurile:
   a) dacă unul dintre operanzi este de tip `byte` şi celălalt este de tip `short`, atunci tipul expresiei condiţionale este `short`;
   b) dacă unul dintre operanzi este de tip T, unde T poate fi `byte`, `short` sau `char`, şi celălalt operand este o expresie constantă de tip `int` a cărei valoare este reprezentabilă în tipul T, atunci tipul expresiei condiţionale este T;
   c) altfel, se fac conversii implicite la compilare tipurilor celor de-al doilea şi al treilea operand şi acesta va fi tipul returnat de expresia condiţională.
3. dacă cel de-al doilea sau al treilea operand este de tip `null` şi tipul celuilalt operand este referinţă, atunci tipul expresiei condiţionale este acel tip referinţă;

4. dacă cel de-al doilea şi al treilea operand sunt tipuri referinţă diferite, atunci se încearcă o conversie de atribuire (engl. *assignment conversion*) a unui tip la celălalt tip. În caz de succes, acesta va fi tipul expresiei condiţionale, altfel se va obţine o eroare de compilare.

Exemplul 2.5.16. Fie declaraţia `byte b=25;`. Tipul întors de expresia condiţională `(1<2)?b:40` este `byte`, deoarece se aplică regula 2.b) de mai sus. Expresia `(1<2)?"abc":40` va returna o eroare la compilare, deoarece nu este posibilă o conversie de atribuire între `String` şi `int`.

Continuăm cu prezentarea succintă a conversiilor numerice. Tipul unei expresii cast este tipul al cărui nume apare între paranteze (se mai numeşte *operator cast*). Rezultatul unei expresii cast nu este o variabilă, ci o valoare, chiar dacă rezultatul expresiei operand este o variabilă. De exemplu, o expresie cast converteşte la execuţie o valoare de tip numeric la o valoare similară a altui tip numeric. Prezentarea în detaliu a operatorului de conversie va fi făcută în secţiunea 2.8 („Conversii").

Exemplul 2.5.17. Fie subprogramul Java:

```
short sh1 = 5;
int in1 = sh1;
System.out.println(in1);
int in2 = 20000;
short sh2 = (short)in2;
System.out.println(sh2);
```

Se observă că de la tipuri „mai mici" (care se reprezintă pe mai puţini octeţi) la tipuri „mai mari" nu este necesară prezenţa conversiei explicite (cum este cazul conversiei de la `short` la `int`). În schimb, dacă se încearcă atribuirea unei variabile de tip mai mare unei variabile de tip mai mic, atunci se obţine eroare la compilare care semnalează pierderea posibilă a unor date (engl. *possible loss of data*). Astfel, este necesară prezenţa conversiei explicite care în situaţia exemplului nostru este `(short)`. Deşi nu mai obţinem eroare la compilare, la execuţie se pot obţine rezultate eronate (datorită trunchierii datelor prin conversia explicită). De exemplu, execuţia subprogramului:

```
int in2 = 200000;
short sh2 = (short)in2;
System.out.println(sh2);
```

va afişa valoarea `3392` obţinută prin „înlăturarea" primilor doi octeţi care reprezintă în memorie valoarea 200000 (3392 reprezintă, de fapt, restul împărţirii lui 200000 la 32768).

Continuăm cu ultimul operator referitor la tipuri intregrale, şi anume operatorul de concatenare a şirurilor. Orice tip Java poate fi convertit la tipul `String`. O valoare x a tipului primitiv T este mai întâi convertită la o valoare referinţă apelând la clasele corespunzătoare tipului T (engl. *wrapper*) având ca parametru pe `x`:

1. **pentru** T = `char`, se foloseşte `new Character(x)`
2. **pentru** T = `byte`, `short`, `int`, se foloseşte `new Integer(x)`
3. **pentru** T = `long`, se foloseşte `new Long(x)`

Pentru obiecte primitive, implementarea poate optimiza crearea unui obiect *wrapper* prin convertirea directă a unui tip primitiv la un obiect de tip `String`.

Exemplul 2.5.18. De exemplu, dorim să citim dintr-un şir de caractere (numit `sir`) un întreg (numit `numar`):

```
Integer obiectIntreg = new Integer(sir);
int numar = obiectIntreg.intValue();
```

Am utilizat unul dintre constructorii clasei `Integer` şi am creat un obiect din această clasă. Metoda `intValue()` va întoarce valoarea întreagă conţinută de `obiectIntreg`. Dacă şirul de caractere nu conţine o valoare numerică, se va arunca excepţia `NumberFormatException`.

Deoarece clasa `Integer` conţine metoda:

```
public static int parseInt(String s);
```

se poate evita crearea unui obiect din clasa `Integer`, astfel:

```
int numar = Integer.parseInt(sir);
```

Implementarea poate alege între conversie şi concatenare într-un pas pentru evitarea creării unui obiect `String` intermediar. De altfel, implementarea Java foloseşte clasa `StringBuffer` pentru a reduce numărul de obiecte `String` create pentru evaluarea unei expresii. Operatorul + este stâng-asociativ, deci `a + b + c` este privit ca `(a + b) + c`.

Exemplul 2.5.19. Fie subprogramul Java, care pune în evidenţă asociativitatea operatorului +:

```
String s1 = "a" + 2 + 3, s2 = 2 + 3 + "b";
System.out.println("s1 = " + s1 + ", s2 = " + s2);
```

În cazul evaluării lui `s1`, ambii operatori + se referă la obiecte `String` şi astfel se convertesc valorile numerice `2`, respectiv `3`, la `"2"`, respectiv `"3"`. În concluzie, `s1` va fi evaluat la `"a23"`. În ceea ce-l priveşte pe `s2`, primul operator + se referă la adunarea a două numere întregi, iar al doilea operator + se referă la concatenarea şirurilor având al doilea argument din clasa `String`. Astfel, `s2` se va evalua la `"5b"` (şi nu la `"23b"`).

Continuăm cu operatorii care lucrează cu operanzi de tip virgulă flotantă. În afara constructorilor şi metodelor din clasele `Float`, `Double` şi `Math`, putem utiliza următorii operatori:

1. *Operatorii de comparaţie*, care întorc o valoare de tip `boolean`: <, <=, >, >=, ==, `!=`
2. *Operatorii numerici*, care întorc o valoare de tip `float` sau `double`: +, - (unari şi binari), `*`, `/`, `%` (multiplicativi), ++, -- (incrementare şi decrementare, formele prefixată şi postfixată)

3. *Operatorul condiţional* `? :`
4. *Operatorul de conversie explicită*, care poate converti o valoare în virgulă mobilă la o valoare de un tip numeric specificat
5. *Operatorul de concatenare a şirurilor*, care în cazul unui operand de tip `String` şi al unuia de tip în virgulă flotantă, va face conversia operandului în virgulă mobilă la `String`, creând astfel un nou obiect `String` egal cu concatenarea acestora.

Dacă un operator numeric în virgulă mobilă are un operand de tip `double`, atunci operaţia este efectuată folosind aritmetica în virgulă mobilă şi 64 de biţi şi convertind celălalt operand la tipul `double`, valoarea returnată de operatorul numeric fiind tot de tip `double`. În caz contrar, operaţia este efectuată folosind o precizie de 32 de biţi, convertind operanzii la tipul `float`, şi astfel rezultatul operatorului numeric este de tip `float`. Operatorii în virgulă flotantă nu indică depăşirea domeniului şi nici nu generează excepţii.

Referitor la operatorii în virgulă flotantă, începem cu operatorii de comparaţie. În plus faţă de operatorii integrali, putem spune că rezultatul unei comparaţii este următorul:

1. Dacă unul dintre operanzi este `NaN`, atunci rezultatul este `false`;
2. Toate valorile diferite de `NaN` sunt ordonate, cu minus infinit mai mic decât toate valorile finite şi plus infinit mai mare decât toate valorile finite;
3. Zero negativ şi pozitiv sunt considerate egale.

Exemplul 2.5.20. Expresia `0.0<0.0` se evaluează la `false`, iar `-0.0<=0.0` se evaluează la `true`. Clasa `Math` conţine două metode statice care întorc minimumul, respectiv maximumul dintre două numere în virgulă flotantă:

```
public static double min(double a, double b);
public static double max(double a, double b);
```

Astfel, execuţia secvenţei de instrucţiuni Java:

```
System.out.println(Math.min(-0.0, 0.0));
System.out.println(Math.max(-0.0, 0.0));
```

va afişa:

```
-0.0
0.0
```

Referitor la operatorii în virgulă flotantă == şi `!=`, distingem în plus faţă de aceiaşi operatori integrali următoarele (conform standardului american IEEE 754):

1. Dacă un operand este `NaN`, atunci rezultatul lui == este `false`, iar rezultatul lui != este `true`;
2. Zero pozitiv şi zero negativ sunt egale;
3. Altfel, două valori distincte în virgulă mobilă se consideră inegale, deci rezultatul lui == este `false`.

Putem spune că valoarea întoarsă de expresia `x != x` este `true` dacă şi numai dacă `x` este `NaN`. Metoda `isNaN()` din clasele `Float` şi `Double` se poate utiliza pentru prevenirea operaţiilor fără sens (care conduc la obţinerea lui `NaN`).

Exemplul 2.5.21. Valoarea întoarsă de expresia `-0.0 == 0.0` este `true`. La fel şi expresia `Double.POSITIVE_INFINITY != Double.NEGATIVE_INFINITY` întoarce `true`.

Referitor la operatorii +, - unari, putem spune în plus că la operanzii în virgulă flotantă negaţia nu mai este la fel ca scăderea din zero. Iată câteva cazuri speciale pentru operatorul unar:

1. dacă operandul este `NaN`, atunci rezultatul este `NaN`;
2. dacă operandul este plus sau minus infinit, atunci rezultatul este minus sau respectiv plus infinit;
3. dacă operandul este zero negativ sau pozitiv, atunci rezultatul va fi zero pozitiv sau respectiv negativ.

Exemplul 2.5.22. Execuţia următorului subprogram Java:

```
double x = +0.0;
System.out.println(0.0-x + " " + -x);
double y = -0.0;
System.out.println(0.0-y + " " + -y);
```

va afişa:

```
0.0 -0.0
0.0 0.0
```

Continuăm cu operatorii +, - binari. Adunarea în virgulă flotantă nu este asociativă. Rezultatul unei adunări în virgulă flotantă se determină după următoarele reguli:

1. Dacă unul dintre operanzi este `NaN`, atunci rezultatul este `NaN`;
2. Suma a două infinităţi de semn opus este `NaN` (operaţie fără sens);
3. Suma a două infinităţi de acelaşi semn este respectivul infinit;
4. Suma dintre un infinit şi o valoare finită este respectivul infinit;
5. Suma a două zerouri de semn opus este zero pozitiv;
6. Suma a două zerouri de acelaşi semn este zero de acel semn;
7. Suma dintre zero şi o valoare finită este acea valoare finită;
8. Suma dintre două numere finite opuse ca semn, dar egale în valoare absolută, este zero pozitiv;
9. În restul cazurilor, se calculează suma. Dacă suma a două numere este prea mare, atunci se întoarce infinit (spre deosebire de numerele întregi). Dacă suma a două numere este prea mică pentru reprezentare, se întoarce zero apropiat ca semn. Altfel, suma este rotunjită către cea mai apropiată valoare reprezentabilă.

Exemplul 2.5.23. Neasociativitatea numerelor în virgulă flotantă se poate observa în următorul subprogram Java, unde suma a două numere în virgulă flotantă depăşeşte domeniul:

```
float a = 2E38f, b =  -a;
System.out.println((a + a) + b);
System.out.println(a + (a + b));
```

care va afişa:

```
Infinity
2.0E38
```

Ca şi la numere întregi, scăderea a două numere în virgulă flotantă este ca o adunare cu opusul numărului. Cu toate acestea, scăderea din zero nu este identică cu negaţia (exemplul 2.5.22). Operaţiile de adunare/scădere nu vor arunca niciodată excepţii de execuţie.

Continuăm cu operatorul de multiplicare (înmulţire) `*` pentru numere în virgulă flotantă. Multiplicarea în virgulă flotantă nu este asociativă. Rezultatul unei înmulţiri în virgulă flotantă este dat de regulile:

1. Dacă unul dintre operanzi este `NaN`, atunci rezultatul este `NaN`;
2. Dacă rezultatul nu este `NaN`, atunci semnul rezultatului este pozitiv dacă ambii operanzi au acelaşi semn sau negativ dacă operanzii au semne diferite (*regula semnelor*);
3. Înmulţirea dintre infinit şi zero este `NaN` (operaţie fără sens);
4. Înmulţirea dintre infinit şi o valoare finită este infinit cu semnul determinat cu regula semnelor;
5. În celelalte cazuri, se calculează produsul celor doi operanzi. Dacă produsul este prea mare, atunci se întoarce infinit. Dacă produsul este prea mic, atunci se întoarce zero. Altfel, produsul este rotunjit către cea mai apropiată valoare reprezentabilă.

Operaţia de înmulţire nu va arunca niciodată excepţii în timpul execuţiei programelor Java.

Continuăm cu operatorul de împărţire, `/`, pentru numere în virgulă flotantă. Împărţirea în virgulă flotantă nu este asociativă. Rezultatul unei împărţiri în virgulă flotantă este dat de regulile:

1. Dacă unul dintre operanzi este `NaN`, atunci rezultatul este `NaN`;
2. Dacă rezultatul nu este `NaN`, atunci semnul rezultatului este pozitiv dacă ambii operanzi au acelaşi semn sau negativ dacă operanzii au semne diferite (*regula semnelor*);
3. Împărţirea dintre infinit şi infinit este `NaN` (operaţie fără sens);
4. Împărţirea dintre infinit şi o valoare finită este infinit cu semnul determinat cu regula semnelor;
5. Împărţirea dintre o valoare finită şi infinit este zero cu semnul determinat cu regula semnelor;

6. Împărţirea dintre o valoare finită şi zero este semnalizată prin aruncarea la execuţie a unei excepţii;
7. În celelalte cazuri, se calculează raportul celor doi operanzi. Dacă raportul este prea mare, atunci se întoarce infinit. Dacă raportul este prea mic, atunci se întoarce zero. Altfel, raportul este rotunjit către cea mai apropiată valoare reprezentabilă.

În cazul împărţirii cu zero se va arunca excepţia `ArithmeticException` în timpul execuţiei programelor Java. Exemplele 2.4.3, 2.4.4, 2.4.5 şi 2.4.6 se referă la operaţiile de înmulţire/împărţire pentru numere în virgulă flotantă.

Continuăm cu prezentarea operatorului `%` pentru operanzi în virgulă flotantă. Prin definiţie, `a%b=a-b*[a/b]`, unde `[x]` reprezintă partea întreagă inferioară a lui `x`. În schimb, regulile IEEE 754 diferă prin considerarea celui mai apropiat întreg în locul părţii întregi. Astfel, `Math.IEEEremainder(a,b)=a-b*Math.rint(a/b)`. Rezultatul unei operaţii `%` în virgulă flotantă este determinat de regulile:

1. Dacă unul dintre operanzi este `NaN`, atunci rezultatul este `NaN`;
2. Dacă rezultatul nu este `NaN`, atunci semnul rezultatului este egal cu semnul operandului care se împarte;
3. Dacă operandul care se împarte este infinit sau operandul la care se împarte este zero, atunci rezultatul este `NaN`;
4. Dacă operandul care se împarte este finit şi operandul la care se împarte este infinit, atunci rezultatul este operandul care se împarte;
5. Dacă operandul care se împarte este zero şi operandul la care se împarte este finit, atunci rezultatul este operandul care se împarte;
6. În celelalte cazuri, se foloseşte formula de mai sus `(a%b=a-b*[a/b])`.

Evaluarea operatorului `%` pentru operanzi în virgulă mobilă nu aruncă niciodată o excepţie la execuţie, chiar dacă operandul din dreapta este zero.

Exemplul 2.5.24. Următorul subprogram Java pune în evidenţă asemănarea şi deosebirea dintre comportarea lui `%` şi a lui `Math.IEEEremainder()`:

```
System.out.println(9.8 % 4.8);
System.out.println(3.2 % 4.8);
System.out.println(9.8 - 4.8 * Math.floor(9.8 / 4.8));
System.out.println(3.2 - 4.8 * Math.floor(3.2 / 4.8));
System.out.println(Math.IEEEremainder(9.8, 4.8));
System.out.println(Math.IEEEremainder(3.2, 4.8));
System.out.println(9.8 - 4.8 * Math.rint(9.8 / 4.8));
System.out.println(3.2 - 4.8 * Math.rint(3.2 / 4.8));
```

va afişa la execuţie:

```
0.20000000000000107
3.2
0.20000000000000107
```

```
3.2
0.20000000000000107
-1.5999999999999996
0.20000000000000107
-1.5999999999999996
```

Operatorii ++ şi -- pentru operanzii în virgulă mobilă sunt similari cu cei integrali.

Exemplul 2.5.25. Următorul subprogram Java:

```
double i = 3.52, j = 7.84, k = ++i + j--;
System.out.println(k);
```

va afişa `12.36`, iar ca efect lateral `i` devine `4.52` şi `j` devine `6.84`.

În cazul operatorului condiţional, care poate avea al doilea şi al treilea operand de tip `float` sau `double`, lucrurile sunt similare cu operanzii integrali.

Exemplul 2.5.26. Fie subprogramul Java:

```
double d = 2.35;
int i = 7;
System.out.println(4<3?d:i);
```

Se va afişa `7.0`, şi nu `7`. Tipul rezultatului expresiei condiţionale trebuie să fie complet determinat în momentul compilării şi, folosind regulile conversiei implicite la compilare, acest tip este `double`. Aşadar, valoarea este tipărită în formatul în virgulă flotantă.

Operatorul de conversie explicită pentru numere în virgulă mobilă este similar cu cel de la numere integrale.

Exemplul 2.5.27. Execuţia subprogramului Java de mai jos:

```
int i1 = 5;
float f1 = i1;
System.out.println(f1);
float f2 = 23.56f;
int i2 = (int)f2;
System.out.println(i2);
```

va afişa:

```
5.0
23
```

Continuăm cu ultimul operator referitor la tipuri în virgulă flotantă, şi anume operatorul de concatenare şiruri. Similar cu operatorul + pentru tipuri integrale, tipurile `float` şi `double` pot fi convertite la tipul `String`. O valoare `x` a tipului primitiv T este mai întâi convertită la o valoare referinţă apelând la clasele corespunzătoare tipului T (engl. *wrapper*) având ca parametru pe `x`:

1. pentru T = `float` se utilizează `new Float(x)`
2. pentru T = `double` se utilizează `new Double(x)`

Pentru obiecte primitive, implementarea poate optimiza crearea unui obiect *wrapper* prin convertirea directă a unui tip primitiv la un obiect de tip `String`. Citirea unui număr în virgulă flotantă se realizează în mod analog ca la tipurile integrale (exemplul 2.5.18). Similară cu metoda `parseInt()` este metoda `parseFloat()` pentru numere în virgulă flotantă.

Exemplul 2.5.28. Următorul cod Java inițializează variabila `val` la o valoare în virgulă mobilă memorată în variabila `sir` de tip `String`:

```
float val = Float.parseFloat(sir);
```

Pentru variabile de tip `double`, se procedează similar ca la `float`.

Operatorul de concatenare + dintre un şir de caractere şi un număr în virgulă flotantă este similar cu operatorul de concatenare dintre un şir şi un număr integral. În continuare, prezentăm operatorii care lucrează cu operanzi de tip `boolean`:

1. *Operatorii relaționali* `==` şi `!=`
2. *Operatorul de complement logic* `!`
3. *Operatorii logici* (şi, sau exclusiv, sau inclusiv): `&`, `^`, `|`
4. *Operatorii condiționali* (şi, sau): `&&`, `||`
5. *Operatorul condițional* `? :`
6. *Operatorul de concatenare şiruri* care, în cazul unui operand de tip `String` şi al unuia de tip `boolean`, va face conversia operandului `boolean` la `String`, creând astfel un nou obiect `String` egal cu concatenarea acestora.

Expresiile booleene determină controlul fluxului în instrucțiunile `if`, `while`, `do`, `for`, precum şi pe cel al operatorului condițional. Primul operand al operatorului condițional poate fi doar expresie booleană.

Operatorii de egalitate booleeni sunt asociativi. Rezultatul evaluării expresiei `a == b` este `true` dacă operanzii `a` şi `b` sunt egali. În caz contrar, rezultatul întors este `false`. Rezultatul evaluării expresiei `a != b` este `true` dacă operanzii `a` şi `b` sunt diferiți (unul este `true` şi celălalt `false`). În caz contrar, rezultatul întors este `false`.

Tipul expresiei operand al operatorului unar `!` trebuie să fie `boolean`, altfel apare o eroare de compilare şi tipul întors de expresia operatorului de complement logic unar este `boolean`. La execuție, valoarea expresiei `!a` este `true`, dacă operandul `a` se evaluează la `false`, şi `false`, dacă operandul `a` se evaluează la `true`.

În cazul operatorilor logici în care operanzii sunt de tip `boolean`, lucrurile sunt foarte simple. Tipul întors de expresiile `a&b`, `a^b` şi `a|b` este `boolean`. Semantic, avem:

- `a&b` întoarce valoarea `true`, dacă valorile lui `a` şi `b` sunt `true`; altfel, rezultatul este `false`;
- `a^b` întoarce valoarea `true`, dacă valorile lui `a` şi `b` sunt distincte; altfel, rezultatul este `false`;
- `a|b` întoarce valoarea `false`, dacă valorile lui `a` şi `b` sunt `false`; altfel, rezultatul este `true`.

Operatorii `&&` şi `||` sunt variante performante (şi „leneşe" în acelaşi timp) ale operatorilor `&` şi `|`. Astfel, operatorul `&&` este echivalent cu `&`, cu deosebirea că evaluează operandul drept numai dacă valoarea operandului din stânga este `true`. Este stâng-asociativ şi fiecare operand al lui `&&` trebuie să fie de tip `boolean`, altfel apare o eroare de compilare. Tipul returnat de expresia dată de operatorul condiţional `&&` este `boolean`. Dacă valoarea operandului `a` din expresia `a&&b` este `false`, atunci operandul `b` nu se mai evaluează, valoarea întoarsă de expresia `a&&b` fiind `false`.

Similar, operatorul `||` este echivalent cu `|`, cu deosebirea că evaluează operandul drept numai dacă valoarea operandului din stânga este `false`. Este stâng-asociativ şi fiecare operand al lui `||` trebuie să fie de tip `boolean`, altfel apare o eroare de compilare. Tipul returnat este `boolean`. Dacă valoarea operandului `a` din expresia `a||b` este `true`, atunci operandul `b` nu se mai evaluează, valoarea întoarsă de expresia `a||b` fiind `true`.

Exemplul 2.5.29. Execuţia următorului subprogram Java:

```
int x = 2;
System.out.println((4 < 3) && (x++ < 4));
System.out.println(x);
```

va afişa:

```
false
2
```

Deoarece operandul stâng `(4<3)` este evaluat la `false`, nu se mai evaluează operandul din dreapta operatorului leneş `&&` care ar fi condus, ca efect lateral, la incrementarea lui `x`. Schimbând ordinea celor doi operanzi, se obţine valoarea `3` pentru `x`:

```
System.out.println((x++ < 4) && (4 < 3));
```

Operatorul condiţional `? :`, în care operanzii sunt booleeni este similar cu cel în care operanzii sunt de tip numeric (mai puţin primul). În expresia `a?b:c`, dacă operandul `b` este de tip `boolean`, atunci şi `c` este de tip `boolean` şi reciproc.

Exemplul 2.5.30. Expresia condiţională `(1<2)?5:(3<4)` va conduce la o eroare de compilare, deoarece nu poate converti o valoare numerică la una booleană.

Continuăm cu ultimul operator referitor la tipul `boolean`, şi anume operatorul de concatenare a şirurilor. Orice tip Java poate fi convertit la tipul `String`. O valoare `x` a tipului primitiv `boolean` este mai întâi convertită la o valoare referinţă apelând la clasele corespunzătoare tipului `boolean` având ca parametru pe `x` astfel: se foloseşte `new Boolean(x)`.

Exemplul 2.5.31. Folosind un constructor al clasei `Boolean` şi rulând programul:

```
Boolean b = new Boolean(1<2);
System.out.println("adevarat ca " + b);
```

se va afişa mesajul „`adevarat ca true`". Acest cod se poate rescrie transparent pentru programator astfel:

```
System.out.println("adevarat ca " + (1<2));
```

Expresiile primare pentru tipul referinţă sunt :

1. `this`
2. *Crearea unei instanţe a unei clase* (folosind cuvântul rezervat `new`);
3. *Accesul la câmpurile unui obiect*, prin nume calificat sau expresie de acces la câmp;
4. *Apelul metodelor*.

Operatorii care lucrează cu referinţele obiectelor sunt:

1. *Operatorul de conversie explicită*
2. *Operatorul de concatenare a şirurilor* +, care în cazul unui operand de tip `String` şi al unuia de referinţă, va face conversia operandului referinţă prin apelul metodei `toString()`, creând în final un nou obiect `String` egal cu concatenarea acestora
3. *Operatorul* `instanceof`
4. *Operatorii de egalitate de referinţă* == şi !=
5. *Operatorul condiţional* ? :

Atât expresiile primare, cât şi operatorii care lucrează cu tipul referinţă vor fi prezentaţi în capitolul 3. Tot în categoria expresiilor primare intră şi accesul la elementele unui tablou (tablourile vor fi prezentate tot în capitolul 3).

2.6. Variabile

O *variabilă* este o locaţie de memorie care are un tip asociat (se mai numeşte tip verificat la compilare). O variabilă conţine o valoare care este compatibilă cu tipul variabilei. Valoarea unei variabile se poate schimba în urma unei asignări sau a folosirii operatorilor ++ şi --. Compatibilitatea valorilor implicite şi toate asignările unei variabile sunt verificate în timpul compilării, cu excepţia tabloului, în cazul căruia verificarea se face în timpul execuţiei programului Java.

Formal, o variabilă este un 5-uplu de forma:

Nume	Tip	Listă_modificatori	Valoare	Adresă

unde *Nume* este un identificator, *Tip* reprezintă tipul variabilei şi este prezentat în secţiunea 2.4, *Listă_Modificatori* reprezintă lista modificatorilor unei variabile, *Valoare* reprezintă valoarea variabilei şi *Adresă* reprezintă adresa acesteia. Modificatorii de acces Java sunt: `public`, `protected`, implicit (engl. *friendly*), `private`, iar ceilalţi (care nu sunt de acces) sunt `final`, `abstract`, `static`, `native`, `transient`, `synchronized` şi `volatile`. Vom reveni asupra acestor modificatori în capitolele următoare. Fiecare variabilă are o adresă, variabilele primitive păstrând valoarea lor în stiva de valori (engl. *Stack Memory*), iar variabilele referință (obiectele și tablourile) își păstrează o referință în stiva de valori, iar valoarea obiectului în ansamblul de valori (engl. *Heap Memory*).

O variabilă de tip primitiv poate ține o valoare care este de acel tip primitiv. O valoare de tip referință poate ține o referință `null` sau o referință la orice obiect a cărui clasă este compatibilă cu tipul variabilei. Cu alte cuvinte, o variabilă a clasei C poate ține o referință `null` sau o referință la o instanță a clasei C sau a unei subclase a lui C. O variabilă de tip interfață poate ține o referință `null` sau o referință la orice instanță a oricărei clase care implementează acea interfață. Dacă T este un tip primitiv, atunci o variabilă de tip tablou de T-uri poate ține o referință `null` sau o referință la orice tablou de T-uri. Dacă R este un tip referință, atunci o variabilă de tip tablou de R-uri poate ține o referință `null` sau o referință la orice tablou de tip tablou de S-uri, unde tipul S se poate converti la tipul R. O variabilă de tip `Object` poate ține o referință `null` sau o referință la orice obiect, indiferent dacă este instanță a unei clase sau tablou.

Exemplu 2.6.1. Pentru a obține o reprezentare a adresei interne a unei variabile obiect vom apela metoda `hashCode()` din clasa `Object`:

```
public int hashCode();
```

care returnează valoarea codului *hash* a unui obiect. De fiecare dată când este apelată pentru același obiect în timpul execuției unui program, metoda `hashCode()` returnează același întreg. Mai mult, dacă două obiecte sunt egale conform cu metoda `equals()`, atunci apelul metodei `hashCode()` aplicate acestor două obiecte furnizează același întreg. Acest întreg nu rămâne însă același de la o execuție la alta a aceluiași program.

Pentru a obține numele clasei care a instanțiat obiectul, vom folosi metoda `getClass()` din clasa `Class`:

```
public final Class getClass();
```

care returnează clasa obiectului în momentul execuției. Această metodă se poate folosi împreună cu metoda `getName()` din clasa `Class`:

```
public String getName();
```

care returnează numele complet al tipului (clasei, interfeței, șirului sau primitivei) reprezentat de obiectul de tip `Class`.

```
public class AdreseVariabile {
  public static void main(String args[]) {
    AdreseVariabile x = new AdreseVariabile();
    System.out.println(
      "Tipul lui x = " + x.getClass().getName() +
      "\n\t valoarea lui x = " + x +
      "\n\t adresa lui x = " + x.hashCode());
    byte [] y = new byte[10];
    System.out.println(
      "Tipul lui y = " + y.getClass().getName() +
      "\n\t valoarea lui y[0] = " + y[0] +
      "\n\t adresa y = " + y.hashCode());
  }
} // sfarsitul definitiei clasei AdreseVariabile
```

La execuţia acestui program se obţin tipul, valoarea şi adresa din memorie ale variabilelor `x` şi `y`:

```
Tipul lui x = AdreseVariabile
    valoarea lui x = AdreseVariabile@720eeb
    adresa lui x = 7474923
Tipul lui y = [B
    valoarea lui y[0] = 0
    adresa y = 3242435
```

De remarcat că valoarea lui `x` este compusă din numele clasei care l-a instanţiat urmat de adresa variabilei în baza 16 separată prin caracterul `'@'`. Cu alte cuvinte, `720eeb`$_{(16)}$ = `7474923`$_{(10)}$. Pentru variabile de tip tablou, numele clasei este codificat prin caracterul `'['` (care sugerează ideea de index) şi, în general, prima literă a clasei *wrapper* asociate (`B` pentru `byte`, excepţii fiind `Z` pentru `boolean`, `J` pentru `long`, `L` urmat de numele clasei sau interfeţei pentru variabile clasă, respectiv interfaţă).

Exemplul 2.6.2. Pentru declaraţia următoare:

```
public static final int k = 4;
```

putem spune că variabila (atribut al unei clase) `k` este de tip `int` şi că este iniţializată cu valoarea `4`. În plus, modificatorii de acces `public`, `static` şi `final` precizează că variabila `k` este publică (accesibilă de oriunde), statică (variabilă de clasă) şi constantă (nu i se poate modifica valoarea). Evident, ordinea de apariţie a modificatorilor `public`, `static` şi `final` se poate schimba.

În Java, există şapte tipuri de variabile:

1. *Variabila clasă* este un atribut (câmp) într-o declaraţie de clasă folosind cuvântul rezervat `static` sau într-o declaraţie de interfaţă (cu sau fără cuvântul rezervat `static`). O variabilă clasă este creată când este încărcată clasa (deci nu neapărat la declararea unei instanţe a clasei) şi este iniţializată cu valoarea implicită. Variabila clasă îşi încetează existenţa când este descărcată clasa sau interfaţa, după ce finalizarea clasei sau interfeţei este completă.
2. *Variabila instanţă* este un atribut dintr-o declaraţie de clasă fără folosirea cuvântului rezervat `static`. Dacă clasa C are un atribut A care este o variabilă instanţă, atunci se va crea şi iniţializa cu valoare implicită o nouă instanţă a variabilei A corespunzătoare fiecărui obiect nou creat al clasei C sau oricărei subclase a clasei C. Variabila instanţă îşi încetează existenţa când obiectul corespunzător atributului nu mai este referenţiat.

3. *Componentele unui tablou* sunt variabile (fără nume) create şi iniţializate cu valori implicite de fiecare dată când un nou obiect tablou este creat. Componentele tabloului îşi încetează existenţa când tabloul nu mai este referenţiat.
4. *Variabila parametru a unei metode* este valoarea argumentului trimis către o metodă. Pentru fiecare parametru din declaraţia metodei, în momentul apelului metodei se creează o nouă variabilă parametru care este iniţializată cu valoarea argumentului corespunzător apelului metodei (în cazul variabilelor de tip primitiv se consideră valoarea acestora, iar în cazul variabilelor tablou şi referinţă se consideră adresa acestora). Variabila parametru îşi încetează existenţa când se termină execuţia metodei respective.
5. *Variabila parametru a unui constructor* este valoarea argumentului trimis către un constructor. Pentru fiecare parametru din declaraţia constructorului, se creează o variabilă parametru de fiecare dată când se apelează implicit sau explicit constructorul. Noua variabilă este iniţializată cu valoarea argumentului corespunzător al apelului constructorului. Variabila parametru a constructorului îşi încetează existenţa când se termină execuţia constructorului respectiv.
6. *Variabila parametru de tratare a excepţiilor* este creată de fiecare dată când este prinsă o excepţie cu o clauză `catch` a unei instrucţiuni `try`. Noua variabilă este iniţializată cu obiectul actual asociat cu acea excepţie. Variabila parametru de tratare a excepţiilor îşi încetează existenţa când se termină execuţia blocului asociat clauzei `catch` respective.
7. *Variabila locală* este declarată de o instrucţiune de declarare a variabilei locale. De fiecare dată când fluxul de control intră într-un bloc sau instrucţiune `for`, se creează o nouă variabilă pentru fiecare variabilă locală declarată. Spre deosebire de celelalte categorii de variabile, variabilele locale nu sunt iniţializate implicit, deci se va depista eroare la compilare dacă se încearcă accesarea unei variabile locale care nu a fost iniţializată explicit. Variabila locală îşi încetează existenţa când se termină execuţia blocului sau a instrucţiunii `for` asociate.

Exemplul 2.6.3. Următorul program Java pune în evidenţă şase categorii de variabile care pot apărea. Un exemplu de folosire a variabilelor de tip tablou este 2.4.9. Menţionăm că variabilele de tip tablou sunt prezentate în capitolul 3:

```
import java.io.*;

public class CitiriIntregi {
  public static void main(String args[]) {
    int numar = 0;
    BufferedReader in;
    String linie;
    try {
      in = new BufferedReader(
                     new InputStreamReader(System.in), 1);
      while (true) {
        System.out.println("Dati numarul " +
                Radical.numarDeNumere);
          linie = in.readLine();
          try {
              numar = Integer.parseInt(linie);
          }
          catch (NumberFormatException f) {
              System.out.println("Nu ati tastat numar intreg");
          }
          Radical obiect = new Radical(numar);
            System.out.println("Radical din " + numar +
              " este " + obiect.radacinaPatrata());
          System.out.println(
              "Continuam ? (tastati s pentru stop)");
          linie = in.readLine();
          if (linie.equals("s") || linie.equals("S"))
               break;
          }
         in.close();
      }
      catch (IOException e) {
       System.err.println("Citire gresita de la tastatura " + e);
      }
   } // de la metoda main()
} // sfarsitul definitiei clasei CitiriIntregi

class Radical {
   static int numarDeNumere = 1;
   int nr;

   Radical (int nr) {
      this.nr = nr;
      numarDeNumere++;
   }

   double radacinaPatrata() {
      return Math.sqrt(nr);
   }
} // sfarsitul definitiei clasei Radical
```

Variabilele utilizate în programul de mai sus sunt:

1. *Variabilă clasă*: `numarDeNumere` este o variabilă statică şi are drept scop numărarea instanţelor clasei `Radical`;
2. *Variabilă instanţă*: `nr` este o variabilă atribut din declaraţia clasei `Radical`;
3. *Componentele unui tablou*: nu avem variabile de tip tablou;
4. *Variabilă parametru al unei metode*: `linie` este argumentul trimis către metoda `parseInt()` din clasa `Integer`;

5. *Variabilă parametru al unui constructor*: `nr` este parametru pentru constructorul clasei `Radical`;
6. *Variabile parametri de tratare a excepţiilor*: `f` şi `e` sunt două variabile care tratează excepţiile `NumberFormatException`, respectiv `IOException`;
7. *Variabile locale*: `numar`, `in`, `linie` sunt variabile locale în metoda `main()`.

Din punctul de vedere al tipului variabilelor, putem spune că `numar`, `numarDeNumere` şi `nr` aparţin tipului primitiv `int`, celelalte variabile fiind de tip referinţă, şi anume: `linie` din clasa `String`, `in` din clasa `BufferedReader`, `obiect` din clasa `Radical`.

2.7. Declaraţii şi iniţializări

Începem cu prezentarea instrucţiunilor de declarare a unei variabile locale. Sintaxa generală este:

```
<Tip> <numeVariabila1> [=<Expresie>] [, <numeVariabila2>
[=<Expresie>]] . . . [, <numeVariabilaN> [=<Expresie>]];
```

unde `<Tip>` se **numeşte tipul variabilelor** al căror nume este `<numeVariabila1>`, `<numeVariabila2>`, ..., `<numeVariabilaN>` (acestea pot fi şi nume de tablouri, caz în care se adaugă paranteze, `[` şi `]`). Parantezele pătrate (`[]`) semnifică variabile de tip tablou (vom vedea în exemplul 2.7.4 că acestea pot apărea atât înaintea, cât şi după numele variabilei). `<Expresie>` reprezintă o expresie de iniţializare care se poate converti la tipul `<Tip>`, altfel obţinându-se eroare la compilare.

Fiecare instrucţiune de declarare a unei variabile locale este conţinută într-un bloc de instrucţiuni. Instrucţiunile de declarare a variabilelor locale pot fi intercalate printre alte instrucţiuni ale blocului. O declarare de variabilă locală poate apărea şi într-o instrucţiune `for`, caz în care se execută la fel ca o instrucţiune de declarare a unei variabile locale.

O instrucţiune de declarare a unei variabile locale este o instrucţiune executabilă. La execuţie, declaratorii sunt procesaţi de la stânga la dreapta. Dacă un declarator are o expresie de iniţializare, atunci expresia este evaluată şi valoarea se asignează variabilei. Dacă declaratorul nu are o expresie de iniţializare, atunci compilatorul Java verifică dacă fiecare referinţă către variabilă este precedată de execuţia unei asignări a variabilei. Dacă nu este aşa, atunci se semnalează o eroare la compilare.

Fiecare iniţializare (cu excepţia celei dintâi) este executată dacă precedenta expresie de iniţializare se evaluează normal. Execuţia unei declaraţii a unei variabile locale se termină normal, dacă evaluarea ultimei expresii de iniţializare decurge normal. Evident, dacă declaraţia unei variabile locale nu conţine expresii de iniţializare, atunci execuţia acesteia se termină întotdeauna normal.

Exemplul 2.7.1. Programul Java de mai jos pune în evidenţă principiile enunţate anterior:

```
class DeclaratiiVariabileLocale {
  public static void main(String[] args) {
    int a, b, c = (a = 1 + (b = 2)) + 1;
    System.out.println("a = " + a + ", b = " + b + ",
     c = " + c);
  }
}
```

Compilatorul va produce ieşirea:

```
a = 3, b = 2, c = 4
```

Deoarece o declaraţie de variabile locale se realizează de la stânga la dreapta, rezultă că se vor declara variabilele locale `a`, `b`, `c` (li se vor aloca în memoria Stack câte 4 octeţi la fiecare). Apoi se va evalua expresia de iniţializare, care se poate împărţi în trei subexpresii. Astfel variabila `b` ia valoarea `2`, expresia (`b = 2`) va întoarce valoarea 2, deci variabila `a` va avea valoarea `3`. Apoi, acestei valori i se adaugă `1` şi deci valoarea variabilei `c` este `4`.

Domeniul unei variabile locale declarate într-un bloc este restul blocului, începând cu instrucţiunea de declarare. Numele unui parametru variabilă locală sau parametru excepţie nu poate fi redeclarat ca variabilă locală, în caz contrar apărând eroare la compilare. Cu alte cuvinte, ascunderea numelui unei variabile locale nu este permisă (cum se întâmplă în cazul variabilelor instanţă ale unei clase). O variabilă locală nu poate fi referită folosind un nume calificat (folosind operatorul de acces „."), ci doar printr-un nume.

Exemplul 2.7.2. Iată un program Java în care punem în evidenţă domeniul de vizibilitate a variabilelor locale:

```
public class DomeniuVariabileLocale {
  static int x = 6;
  public static void main(String[] args) {
    System.out.println("x = " + x);
    int x = (x = 3) * 4;
    System.out.println("x = " + x);
  }
}
```

Ieşirea programului va fi:

```
x = 6
x = 12
```

La prima afişare este vorba de variabila `x` a clasei `DomeniuVariabileLocale` (fiind declarată `static`, este accesibilă dintr-o metodă statică). La cea de-a doua afişare este vorba de variabila locală a metodei `main()` cu acelaşi nume. Domeniul variabilei locale `x` începe imediat după declararea acesteia, deci în expresia `(x = 3)` este vorba de variabila locală, şi nu de variabila clasă.

Domeniul unei variabile locale declarate într-o instrucţiune `for` este în toată instrucţiunea, inclusiv iniţializatorul. Dacă o declaraţie a unui identificator ca variabilă locală apare în domeniul unei variabile parametru sau locale cu acelaşi nume, atunci apare o eroare la compilare.

Exemplul 2.7.3. Ilustrăm acum marginile domeniilor variabilelor:

```
class MarginiDomeniuVariabile {
   public static void main(String[] args) {
      {
        int i = 1;
        System.out.println("i = " + i);
      }
      for (int i = 0; i < 4; i++)
         System.out.println("i = " + i);
   }
}
```

Ieşirea va fi:

```
i = 1
i = 0
i = 1
i = 2
i = 3
```

Prima apariţie a variabilei locale `i` este între cele două acolade, iar cea de-a doua apariţie este în instrucţiunea `for`. Astfel, domeniile acestor două variabile locale cu acelaşi nume sunt disjuncte. Dacă nu ar fi fost prezente cele două acolade, atunci am fi obţinut eroare la compilare.

Fiecare declarator dintr-o declaraţie a unei variabile locale declară o singură variabilă locală, al cărei nume este identificatorul care apare în declarator. Tipul variabilei apare la începutul declarării, urmată eventual de perechi de paranteze pătrate în cazul tablourilor. Tablourile sunt prezentate detaliat în capitolul 4.

Exemplul 2.7.4. Astfel, declaraţia:

```
int m1, m2[], m3[][];
```

este echivalentă cu următoarele trei declaraţii:

```
int m1;
int m2[];
int m3[][];
```

Parantezele pătrate se pot specifica şi la început. De exemplu:

```
int [][] m1, m2[], m3[][];
```

este echivalent cu:

```
int m1[][];
int m2[][][];
int m3[][][][];
```

2.8. Conversii

Fiecare expresie Java are un tip care poate fi determinat din structura expresiei şi tipurile literalilor, variabilelor şi metodelor menţionate în expresie. Sunt cazuri când anumite expresii apar într-un context în care tipul expresiei nu este potrivit, de exemplu, condiţia dintr-o instrucţiune `if` trebuie să aibă tipul `boolean` (şi nu alt tip), altfel apare o eroare la compilare. În anumite cazuri, contextul poate fi capabil să accepte un tip apropiat de tipul expresiei. În loc să fie obligatorie scrierea tipului dorit (ceea ce se mai numeşte *conversie explicită*), Java face o *conversie implicită* a tipului expresiei la un tip acceptabil pentru contextul respectiv.

În fiecare context, sunt permise doar câteva conversii specifice:

- conversii identice;
- conversii primitive implicite (de la mic la mare, engl. *widening conversions*);
- conversii primitive explicite (de la mare la mic, engl. *narrowing conversions*);
- conversii de referinţă implicite;
- conversii de referinţă explicite;
- conversii la `String`.

Se numeşte *conversie identică* o conversie de la un tip la acelaşi tip. Astfel, aceasta implică determinarea tipului dorit pentru expresie şi, în plus, măreşte claritatea programului.

Exemplul 2.8.1. Conversia identică clarifică tipul unui literal. Execuţia secvenţei de cod Java:

```
System.out.println(32768);
System.out.println((short)32768);
```

va afişa:

```
32768
-32768
```

Explicaţia este simplă. Prima apariţie a literalului `32768` va avea asociat tipul `int`, iar a doua apariţie tipul `short`. Cum această din urmă valoare depăşeşte domeniul tipului `short`, va fi afişat `-32768`.

Se numeşte *conversie primitivă implicită* o conversie de la un tip primitiv cu domeniul mai mic la un tip primitiv cu domeniul mai mare, după cum urmează:

- de la `byte` la `short`, `int`, `long`, `float` şi `double`
- de la `short` la `int`, `long`, `float` şi `double`

- de la `char` la `int`, `long`, `float` şi `double`
- de la `int` la `long`, `float` şi `double`
- de la `long` la `float` şi `double`
- de la `float` la `double`

Conversiile primitive implicite nu pierd informaţii despre valoarea numerică iniţială, deoarece se adaugă octeţi la reprezentarea sa. Totuşi, există situaţii când la conversia dintre un tip integral la un tip în virgulă flotantă se poate pierde precizia, adică se pot pierde ultimii biţi nesemnificativi. În acest caz, valoarea în virgulă mobilă se va rotunji la o valoare întreagă apropiată. Justificarea faptului că se poate pierde precizia este foarte simplă: în secţiunea 2.4, am văzut că precizia tipului `float` este de 6-7 cifre semnificative, iar a tipului `double` de 14-15 cifre semnificative. Aşadar, considerând un număr întreg de cel puţin 8 cifre, prin conversie implicită la tipul `float` se vor pierde cel puţin 1-2 cifre (exemplul 2.8.2). În ciuda faptului că se poate pierde precizia, conversiile primitive implicite nu aruncă niciodată excepţii.

Exemplul 2.8.2. Următorul program Java pune în evidenţă faptul că uneori conversia primitivă implicită a două numere întregi mari distincte poate conduce la acelaşi număr de tip `float`:

```
public class PierdePrecizia {
  public static void main(String[] args) {
    int cinci = 555555555;
    float cinciMobil = cinci;
    System.out.println(cinciMobil);
    int cinciSase = 555555566;
    float cinciSaseMobil = cinciSase;
    System.out.println(cinciSaseMobil);
    // afisam diferenta dintre acestea
    System.out.println(cinci - (int)cinciMobil);
    System.out.println(cinciSase - (int)cinciSaseMobil);
  }
}
```

Execuţia programului de mai sus conduce la afişarea datelor:

```
5.5555558E8
5.5555558E8
-29
-18
```

Se numeşte *conversie primitivă explicită* o conversie de la un tip primitiv cu domeniul mai mare la un tip primitiv cu domeniu mai mic, după cum urmează:

- de la `byte` la `char`
- de la `short` la `byte` şi `char`
- de la `char` la `byte` şi `short`
- de la `int` la `byte`, `short` şi `char`

- de la `long` la `byte`, `short`, `char` şi `int`
- de la `float` la `byte`, `short`, `char`, `int` şi `long`
- de la `double` la `byte`, `short`, `char`, `int`, `long` şi `float`

Conversiile primitive explicite pot pierde informaţii despre valoarea numerică iniţială, deoarece se elimină octeţii semnificativi din reprezentarea sa. Atât la conversia de la tipurile integrale, cât şi de la `char`, se păstrează octeţii nesemnificativi (cei din dreapta reprezentării în memorie). În plus, poate să difere şi semnul valorii-rezultat faţă de semnul valorii iniţiale.

Exemplul 2.8.3. Execuţia subprogramului Java de mai jos:

```
char c = '\u00EF';
System.out.println((byte)c + " " + (short)c);
```

va afişa:

```
-17 239
```

deoarece reprezentarea în memorie a lui `c` este:

```
00000000 11101111
```

Aşadar, convertind caracterul `c` la tipul `byte` se va obţine un număr negativ, deoarece bitul de semn al octetului nesemnificativ (al optulea bit de la dreapta la stânga) este `1`. Folosind procedeul de calcul al reprezentării numerelor negative din memorie (exemplul 2.5.13), rezultă că va fi tipărit `-17`. În ceea ce priveşte conversia la tipul `short`, lucrurile sunt mai uşoare, deoarece acesta se reprezintă tot pe doi octeţi şi astfel va fi afişat `239`.

În cazul conversiei explicite a unui număr în virgulă flotantă la un tip integral `<tip>` (unde `<tip>` poate fi `long`, `int`, `char`, `short`, `byte`), există doi paşi:

1. numărul în virgulă flotantă este convertit la `long`, dacă `<tip>` este `long`, sau la `int`, dacă `<tip>` este `byte`, `short`, `char`, `int`, după cum urmează:
   a) dacă numărul în virgulă flotantă este `NaN`, rezultatul este `0` (de tip `long`, respectiv `int`);
   b) altfel, dacă numărul în virgulă flotantă nu este infinit, valoarea în virgulă flotantă este rotunjită la o valoare întreagă de tip `long` sau `int`, după caz;
   c) altfel (numărul în virgulă flotantă este infinit), dacă valoarea este prea mică, atunci rezultatul va fi cea mai mică valoare reprezentabilă de tip `long` sau `int`, iar dacă valoarea este prea mare, atunci rezultatul va fi cea mai mare valoare reprezentabilă de tip `long` sau `int`;
2. dacă `<tip>` este `long` sau `int`, atunci rezultatul conversiei este cel furnizat de primul pas, iar dacă `<tip>` este `char`, `short`, `byte`, atunci se va returna conversia rezultatului primului pas la `<tip>`.

Exemplul 2.8.4. Programul Java de mai jos pune în evidenţă comportarea conversiilor la „capetele" domeniului tipului `float`:

```
public class ConversiiExpliciteFlotante {
  public static void main(String[] args) {
    float min = Float.NEGATIVE_INFINITY;
    float max = Float.POSITIVE_INFINITY;
    System.out.println(
      "long: " + (long) min + " " + (long) max);
    System.out.println("int: " + (int) min + " " + (int) max);
    System.out.println(
      "char:" + (int)(char) min + " " + (int)(char) max);
    System.out.println(
      "short: " + (short) min + " " + (short) max);
    System.out.println(
      "byte: " + (byte) min + " " + (byte) max);
  }
}
```

La execuţie, se va afişa:

```
long: -9223372036854775808 9223372036854775807
int: -2147483648 2147483647
char: 0 65535
short: 0 -1
byte: 0 -1
```

În cazul conversiei la `char`, am mai realizat o conversie explicită la `int` pentru a vedea codurile întregi ale caracterelor respective. În ceea ce priveşte conversiile la tipurile `short` şi `byte`, acestea nu reprezintă minimul, respectiv maximul domeniilor acestora! După cum s-a precizat în cei doi paşi de mai sus, se va face mai întâi conversia la tipul `int` care va implica obţinerea rezultatelor aşteptate: `0x80000000` (minimul), respectiv `0x7fffffff`. Deci, după trunchierea făcută de conversia pentru `short`, se vor obţine valorile: `0x0000` (adică `0`), respectiv `0xffff` (adică `-1`), iar pentru `byte` se vor obţine valorile: `0x00` (adică `0`), respectiv `0xff` (adică `-1`).

Conversiile explicite de la `double` la `float` folosesc modul de rotunjire a valorii în virgulă flotantă. O valoare prea mică a tipului `double` este convertită la zero negativ sau pozitiv, după caz; o valoare prea mare a tipului `double` este convertită la plus/minus infinit, după caz. O valoare `NaN` de tip `double` este convertită la valoarea `NaN` de tip `float`. În ciuda faptului că prin conversii se pot pierde informaţii, acestea nu aruncă excepţii în timpul execuţiei programului.

Exemplul 2.8.5. Execuţia programului Java:

```
public class AlteConversiiExpliciteFlotante {
  public static void main(String[] args) {
    System.out.println(
      "Valoare prea mare float: (int)1e15f==" + (int) 1e15f);
    System.out.println(
      "Float.NaN: (int)NaN==" + (int) Float.NaN);
      System.out.println(
        "Valoare prea mare double: (float)-1e40==" + (float) -1e40);
      System.out.println(
        "Valoare prea mica double: (float)1e-50==" + (float) 1e-50);
    }
}
```

va afişa:

```
Valoare prea mare float: (int)1e15f==2147483647
Float.NaN: (int)NaN==0
Valoare prea mare double: (float)-1e40==-Infinity
Valoare prea mica double: (float)1e-50==0.0
```

Conversiile la tipul referinţă implicite şi explicite sunt prezentate în cadrul capitolului 3.

2.8.1. Contextele de conversie

În Java există cinci *contexte de conversie* în care poate apărea conversia expresiilor Java. Un exemplu frecvent de context de conversie este operandul operatorului numeric +. Procesul de conversie pentru aceşti operanzi se numeşte *conversie numerică la compilare* (engl. *numeric promotion*). De exemplu, `"4" + 5` va implica conversia literalului întreg `5` la literalul `"5"` de tip `String`. Astfel, se va apela operatorul + pentru doi literali de tip `String`, expresia fiind evaluată la valoarea `"45"`.

Cele cinci tipuri de contexte de conversie sunt:

1. *Conversia de asignare* converteşte tipul unei expresii la tipul unei variabile specificate. Aceasta nu aruncă niciodată excepţii.
2. *Conversia la apelul unei metode* se aplică fiecărui argument din apelul unei metode sau constructor şi, în general, face aceleaşi conversii pe care le face conversia de asignare. Aceasta nu aruncă niciodată excepţii.
3. *Conversia explicită* (engl. *cast*) converteşte tipul unei expresii la un tip explicit specificat de operatorul `cast`. Aceasta permite mai multe conversii specifice (cu excepţia celui `String`) şi anumite conversii explicite la un tip referinţă pot cauza aruncarea unei excepţii la execuţie.
4. *Conversia la String* permite oricărui tip să fie convertit la tipul `String`.
5. *Conversia numerică la compilare* (engl. *numeric promotion*) aduce operanzii operatorului numeric la un tip comun pentru efectuarea operaţiei.

Începem cu prezentarea *conversiei de asignare*. Aceasta apare când valoarea unei expresii este asignată unei variabile. Aşadar, tipul expresiei trebuie să fie convertit la tipul variabilei. Contextele de asignare permit folosirea conversiei de identitate, a conversiei primitive implicite şi a conversiei de referinţă implicite. În plus, o conversie primitivă explicită poate fi folosită, dacă au loc condiţiile:

- expresia este o constantă de tip `int`;

- tipul variabilei este `byte`, `short` sau `char`;
- valoarea expresiei este reprezentabilă în tipul variabilei.

Dacă tipul expresiei nu poate fi convertit la tipul variabilei printr-o conversie permisă într-un context de asignare, atunci apare o eroare de compilare. În caz contrar, dacă tipul expresiei poate fi convertit la tipul variabilei printr-o conversie de asignare, atunci se spune că expresia (sau valoarea sa) este *asignabilă* variabilei (sau, echivalent, tipul expresiei este *compatibil la asignare* cu tipul variabilei). O conversie de asignare nu aruncă niciodată excepţii (cu excepţia situaţiei când sunt implicate elementele unui tablou). O valoare a unui tip primitiv nu poate fi asignată unei variabile de tip referinţă, în caz contrar se va obţine o eroare de compilare (contextele de asignare ale tipului referinţă sunt prezentate în capitolul 3). O valoare de tip `boolean` poate fi asignată doar unei variabile de tip `boolean`.

Exemplul 2.8.6. Următorul exemplu pune în evidenţă mai multe contexte de asignare:

```
public class ContextAsignare {
  public static void main(String[] args) {
    // literalul 34 de tip int este convertit la asignare
    // la tipul short
    short s = 34;
    // variabila s este convertita la asignare la tipul float
    float f = s;
    System.out.println("f = " + f);
    char c = '\u0123';
    // variabila l este convertita la asignare la tipul long
    long l = c;
    System.out.println("l = " + l);
    f = 4.55f;
    // variabila f este convertita la asignare la tipul float
    double d = f;
    System.out.println("d = " + d);
  }
}
```

Acesta va afişa la execuţie:

```
f = 34.0
l = 291
d = 4.550000190734863
```

Continuăm cu *conversia la apelul unei metode*. Aceasta se aplică fiecărei valori a argumentelor din apelul unei metode sau al unui constructor. Tipul expresiei argument trebuie să poată fi convertit la tipul parametrului corespunzător. Contextul apelului unei metode permite utilizarea conversiei de identitate, a conversiei primitive şi de referinţă implicite.

Exemplul 2.8.7. Următorul subprogram Java evidenţiază folosirea conversiei la apelul unei metode:

```
float f = 35.7f;
double d = Math.sin(f);
System.out.println("Math.sin(" + f + ") = " + d);
```

Metoda statică `sin()` din clasa `Math` are argumentul de tip `double`, deci în momentul apelului său se va face o conversie a lui `f` la tipul `double`.

Conversiile la apelul unei metode nu includ conversia implicită a constantelor întregi, cum era cazul la conversiile de asignare.

Exemplul 2.8.8. Compilarea programului Java de mai jos:

```
public class ConversieApelMetoda {
  static int f(byte a, int b) { return a + b; }
  static int f(short a, short b) { return a + b; }
  public static void main(String[] args) {
    System.out.println(f(3, 7));
  }
}
```

va conduce la o eroare de compilare. Aceasta, deoarece literalii întregi `3` şi `7` au tipul `int` şi nu găsim nici o metodă `f()` care să aibă ambii parametri de tip `int`. De exemplu, putem rezolva apelul prin conversie explicită:

```
System.out.println(f((short)3, (short)7));
```

sau:

```
System.out.println(f((byte)3, 7));
```

Continuăm cu *conversiile explicite.* Acestea se aplică operandului operatorului `cast`, precedând o valoare cu un nume de tip între paranteze. Astfel, tipul expresiei operandului trebuie convertit la tipul numit explicit de operatorul `cast`. Contextul conversiilor explicite permite folosirea conversiei de identitate, a conversiilor primitive şi de referinţă implicite şi explicite. Anumite conversii explicite conduc la erori de compilare. O valoare a unui tip primitiv poate fi convertită la tipul unei conversii de identitate, la tipul unei conversii primitive implicite sau explicite. O valoare a unui tip primitiv nu poate fi convertită la un tip referinţă prin conversie explicită şi nici invers, adică o valoare a unui tip referinţă nu poate fi convertită explicit la un tip primitiv. Conversiile explicite dintre tipurile referinţă sunt prezentate în capitolul 3.

Exemplul 2.8.9. Următoarea linie de cod Java:

```
int i = 3.46f;
```

conduce la o eroare la compilare, deoarece un literal în virgulă flotantă nu se poate converti implicit la tipul `int`. Trebuie utilizată o conversie explicită, astfel:

```
int i = (int)3.46f;
```

Continuăm cu prezentarea *conversiei la String*. Există conversii la tipul `String` ale oricărui alt tip, inclusiv ale tipului `null`. Conversia la `String` se aplică operanzilor operatorului binar + când unul dintre argumente este `String`. În acest caz special, celălalt argument este convertit la `String`, creându-se un nou `String` care este concatenarea celor două `String`-uri (exemplul 2.5.19).

Conversia numerică la compilare se aplică operanzilor unui operator aritmetic. Contextul conversiei numerice la compilare permite utilizarea conversiei de identitate sau a conversiei primitive implicite. Conversiile numerice la compilare sunt utile la conversia operanzilor unui operator numeric la un tip comun, astfel încât operaţia să se poată efectua. Există două tipuri de conversii numerice la compilare: *unare* şi *binare* (după cum conversia se aplică unui operand sau ambilor operanzi).

În cazul *conversiei numerice unare la compilare*, expresiile operatorilor unari întorc o valoare de tip numeric, după cum urmează:

1. Dacă operandul este de tip `byte`, `short`, `char`, atunci acesta se converteşte la o valoare de tip `int` prin conversie implicită;
2. În caz contrar, operandul rămâne aşa cum este (adică nu i se aplică nici o conversie).

Conversia numerică unară la compilare se execută asupra expresiilor în următoarele situaţii:

1. în expresia de dimensiune a creării unui tablou;
2. în expresia de index a accesului la elementele unui tablou;
3. în cazul operanzilor operatorilor unari + şi –;
4. în cazul operandului operatorului complement faţă de doi;
5. fiecărui operand al operatorilor de deplasare <<, >>, >>> (chiar dacă operandul din dreapta este de tip `long`, acest lucru nu va implica şi conversia primului operand la `long`).

Pentru comportarea operatorilor de deplasare se poate consulta exemplul 2.5.11.

Exemplul 2.8.10. Următorul program Java pune în evidenţă toate categoriile de conversii implicite unare la compilare (vom reveni asupra celor referitoare la tablouri în capitolul 4):

```
public class ConversiiImpliciteUnareLaCompilare {
  public static void main(String[] args) {
    byte b = 2;

    // urmeaza o conversie la compilare pentru dimensiunea
    // unui tablou
    int v[] = new int[b];

    // caracterul c va avea codul 1
    char c = '\u0001';

    // urmeaza o conversie la compilare pentru indexul unui
    // tablou
    v[c] = 1;

    // urmeaza o conversie la compilare pentru minus unar
    v[0] = -c;
   System.out.println("v[0] = " + v[0] + ", v[1] = " + v[1]);

    // asignam -1 lui b. Reprezentarea acestuia in memorie
    // este cu toti bitii 1
    b = -1;

    // urmeaza o conversie la compilare pentru operatorul
    // complement fata de doi
    int i = ~b;
    System.out.println("~0x" + Integer.toBinaryString(b)
      + " = 0x" + Integer.toBinaryString(i));

    // urmeaza o conversie la compilare pentru operatorul
    // de deplasare stanga
    i = b << 4L;
    System.out.println("0x" + Integer.toBinaryString(b)
      + "<<4L =\n\t= 0x" + Integer.toBinaryString(i));
  }
}
```

Acesta va afişa la execuţie:

```
v[0] = 1, v[1] = 1
~0x11111111111111111111111111111111 = 0x0
0x11111111111111111111111111111111<<4L =
    = 0x11111111111111111111111111110000
```

Conversia numerică binară la compilare se aplică unei perechi de operanzi, care trebuie să noteze un tip numeric şi care trebuie să îndeplinească următoarele reguli de conversie implicită a operanzilor:

1. Dacă unul dintre operanzi este de tip `double`, atunci şi celălalt va fi convertit la `double`;
2. Altfel, dacă unul dintre operanzi este de tip `float`, atunci şi celălalt va fi convertit la `float`;
3. Altfel, dacă unul dintre operanzi este de tip `long`, atunci şi celălalt va fi convertit la `long`;
4. Altfel, ambii operanzi vor fi convertiţi la `int`.

Conversia numerică binară la compilare se execută asupra expresiilor în următoarele situaţii:

1. pentru operatorii multiplicativi `*`, `/` şi `%`

2. pentru operatorii de adunare + şi scădere - pentru tipurile numerice
3. pentru operatorii de comparaţie numerici <, <=, >, >=
4. pentru operatorii de egalitate numerici == şi !=
5. pentru operatorii pe biţi care operează pe tipuri integrale &, ^ şi |
6. pentru operatorul condiţional ? : (în anumite cazuri)

Exemplul 2.8.11. Următorul program Java pune în evidenţă câteva dintre categoriile de conversii implicite binare la compilare:

```
public class ConversiiImpliciteBinareLaCompilare {
  public static void main(String[] args) {
    int i = 0;
    float f = 1.0f;
    double d = 2.0;

    // Produsul i * f este convertit la compilare
    // in float * float, apoi testul
    // float == double este convertit
    // in double == double
    if (i * f == d)
      System.out.println("Are loc egalitatea.");

    // O expresie char & byte este convertita la int & int
    byte b = 31; // b are reprezentarea  00011111
    char c = 'G'; // c are reprezentarea 00000000 01000111
    int cSiB = c & b;
    System.out.println(Integer.toBinaryString(cSiB));

    // O expresie a operatorului conditional
    // boolean ? int : float este convertita
    // la compilare boolean ? float : float
    System.out.println((1 < 2) ? i : 4.0f);
  }
}
```

Acesta va afişa la execuţie:

```
111
0.0
```

La afişarea reprezentării binare a valorii variabilei `cSiB`, s-au eliminat zerourile semnificative (în număr de 29). În ceea ce priveşte operatorul condiţional ? :, mai întâi se converteşte variabila `i` la tipul `float`, apoi se întoarce valoarea `0.0` de tip `float` (care se afişează).

2.9. Structuri de control

Execuţia unui program Java este controlată de *structuri de control* sau *instrucţiuni*, care sunt executate pentru efectul lor şi nu pentru faptul că întorc valori (cum era cazul expresiilor, secţiunea 2.5). Anumite instrucţiuni conţin alte instrucţiuni (sau expresii) ca parte a structurii lor. Instrucţiunile Java sunt: instrucţiunea vidă, etichetele, expresiile, `if`, `switch`, `while`, `do`, `for`, `break`, `continue`, `return`, `throw`, `synchronized` şi `try`. Spre deosebire de alte limbaje de programare, în Java nu există instrucţiunea `goto`. Totuşi cuvântul `goto` este rezervat.

Fiecare instrucţiune are un mod normal de execuţie în care sunt făcuţi anumiţi paşi de calcul. Dacă toţi paşii au loc fără vreo terminare anormală, atunci spunem că instrucţiunea *s-a terminat normal*. Cauzele posibile ale terminării anormale ale unei instrucţiuni sunt:

1. instrucţiunile `break`, `continue` şi `return` pot cauza transferul controlului programului prin terminarea anormală a unei instrucţiuni care le conţine;
2. evaluarea anumitor expresii sau instrucţiunile `throw` pot arunca excepţii ale maşinii virtuale Java. O excepţie poate cauza transferul controlului programului prin terminarea anormală a unei instrucţiuni care o conţine.

Motivele pentru care poate apărea o terminare anormală a unei instrucţiuni sunt: `break` (cu sau fără etichetă), `continue` (cu sau fără etichetă), `return` (cu sau fără valoare precizată) şi `throw` (cu valoare precizată, incluzând excepţiile aruncate de maşina virtuală Java). Dacă o instrucţiune evaluează o expresie, terminarea anormală a expresiei va implica şi terminarea anormală a întregii instrucţiuni (din acelaşi motiv). Toţi paşii care ar fi urmat în execuţie nu mai sunt făcuţi.

Înainte de a începe prezentarea sintaxei şi semanticii instrucţiunilor Java, vom descrie noţiunea de *bloc* (de instrucţiuni). Un *bloc* (de instrucţiuni) este o secvenţă de instrucţiuni (pot fi şi instrucţiuni de declarare a variabilelor locale) cuprinsă între acolade. Un bloc este executat prin execuţia fiecărei instrucţiuni a blocului de la prima la ultima şi de la stânga la dreapta.

Declaraţiile variabilelor locale sunt prezentate în secţiunea 2.7. O declaraţie de variabilă locală este o instrucţiune executabilă. La execuţia acesteia, declaratorii de variabile sunt procesaţi de la stânga la dreapta. Dacă un declarator are o expresie de iniţializare, expresia este evaluată şi valoarea este asignată variabilei. Dacă declaratorul unei variabile (să o notăm cu `v`, de exemplu) nu are o expresie de iniţializare, atunci compilatorul Java va verifica dacă fiecare referire la variabila `v` este precedată de o iniţializare a acesteia. În caz contrar, se va obţine o eroare încă de la compilare, primind un mesaj de genul: `variable v might not have been initialized`. Fiecare iniţializare (cu excepţia celei dintâi) se execută doar dacă evaluarea expresiilor de iniţializare precedente se termină normal. Dacă declaraţia de variabile locale nu conţine expresii de iniţializare, atunci execuţia acesteia se termină mereu cu succes.

2.9.1. Instrucţiunea vidă

Instrucţiunea vidă are sintaxa `;` şi nu face nimic. Aceasta se termină normal mereu.

2.9.2. Instrucţiunea etichetă

Instrucţiunea etichetă constă din numele unui identificator, urmat de caracterul două puncte („:"). După caracterul „:" urmează o instrucţiune Java. Instrucţiunile etichetă sunt utilizate împreună cu instrucţiunile `break` şi `continue`. O instrucţiune etichetă nu poate apărea în altă instrucţiune etichetă cu acelaşi identificator, în caz contrar, apare o eroare la compilare. Două instrucţiuni etichetă pot avea acelaşi identificator doar dacă nu sunt incluse una în alta.

Nu există restricţii asupra utilizării aceluiaşi identificator ca etichetă şi ca nume de pachet, clasă, interfaţă, metodă, atribut, parametru sau variabilă locală.

O instrucţiune etichetă se realizează prin execuţia instrucţiunii care urmează. Dacă instrucţiunea este etichetată de un identificator `id` şi instrucţiunea care urmează după „:", notată cu `Instr`, se termină anormal din cauza unui `break` cu acelaşi identificator, atunci spunem că instrucţiunea de etichetă `id` s-a terminat normal. În toate celelalte cazuri de terminare anormală a instrucţiunii `Instr`, spunem că instrucţiunea de etichetă `id` s-a terminat anormal.

Exemplul 2.9.1. Următorul program Java evidenţiază modalitatea de utilizare a instrucţiunilor etichetă (precum şi a instrucţiunilor `for`, `break` etc. care se vor prezenta ulterior):

```
public class InstructiuneEticheta {
  public static void main(String args[]) {
    int max = 5;
    for (int i = 0; i < max; i++) {
      unu: {i--;
            System.out.println("i (unu) = " + i);
            for (int j = 0; j < max; j++)
              if (j == i) break unu;
            System.out.println(
                "S-a executat instructiunea for (unu)");
           }
      doi: {i += 2;
            System.out.println("i (doi) = " + i);
            for (int j = 0; j < max; j++)
              if (j == i) break doi;
            System.out.println(
                "S-a executat instructiunea for (doi)");
           }
```

```
            System.out.println("i = " + i);
        }
    }
}
```

La execuţie se va afişa:

```
i (unu) = -1
S-a executat instructiunea for (unu)
i (doi) = 1
i = 1
i (unu) = 1
i (doi) = 3
i = 3
i (unu) = 3
i (doi) = 5
S-a executat instructiunea for (doi)
i = 5
```

La prima iteraţie a instrucţiunii `for`, valoarea lui `i` este `-1`, deci se va executa instrucţiunea `for` interioară fără a se ieşi necondiţionat. Apoi, se va executa instrucţiunea etichetată cu `doi`, care va implica ieşirea din instrucţiunea `for` interioară când valoarea lui `j` este `1`. După cele trei iteraţii, se va termina execuţia întregului program.

2.9.3. Instrucţiunea expresie

Instrucţiunea expresie are forma generală `<expresie>;`, unde `<expresie>` poate fi o asignare, expresie de preincrementare, predecrementare, postincrementare, postdecrementare, apelul unei metode, expresie de creare a unei instanţe a unei clase. O instrucţiune expresie se execută prin evaluarea sa (adică a lui `<expresie>`).

Dacă aceasta are o valoare, atunci aceasta este returnată. Execuţia unei instrucţiuni expresie se termină normal dacă şi numai dacă evaluarea expresiei se termină normal.

În Java, nu există conversie către `void` (cum era în alte limbaje de programare: C, C++). Aceasta se poate simula prin instrucţiuni de asignare sau prin instrucţiuni de declarare a unei variabile locale.

Instrucţiunile expresie sunt utilizate frecvent. Vezi exemplele 2.5.1, 2.5.2, 2.5.3, 2.5.9, 2.5.25, 2.7.1, 2.7.2.

2.9.4. Instrucţiunea `if`

Instrucţiunea `if` are două forme: simplă (engl. *if-then*) şi completă (engl. *if-then-else*). Instrucţiunea `if` permite execuţia condiţionată a unei instrucţiuni. Sintaxa lui `if-then` este:

```
if (<Expresie>) <Instructiune>
```

iar a lui `if-then-else` este:

```
if (<Expresie>) <Instructiune1> else <Instructiune2>
```

unde `<Expresie>` trebuie să aibă tipul `boolean`, în caz contrar se obţine o eroare de compilare.

În cazul instrucţiunii `if-then` se evaluează `<Expresie>`. Dacă evaluarea se termină anormal, atunci instrucţiunea `if` se termină anormal. Dacă evaluarea lui `<Expresie>` este `true`, atunci se execută `<Instructiune>` şi astfel instrucţiunea `if` se termină normal. Dacă evaluarea lui `<Expresie>` este `false`, atunci nu se execută nimic şi instrucţiunea `if` se termină normal.

În cazul instrucţiunii `if-then-else` se evaluează `<Expresie>`. Dacă evaluarea se termină anormal, atunci instrucţiunea `if` se termină anormal. Dacă evaluarea lui `<Expresie>` este `true`, atunci se execută `<Instructiune1>` şi astfel instrucţiunea `if` se termină normal. Dacă evaluarea lui `<Expresie>` este `false`, atunci se execută `<Instructiune2>` şi instrucţiunea `if` se termină normal.

O situaţie deosebită în cazul instrucţiunii `if` o reprezintă asocierea ramurii `else` în mod corect. Pentru claritate se pot utiliza acolade de delimitare a blocului în care este scrisă instrucţiunea `if`.

Exemplul 2.9.2. Subprogramul Java:

```
int a = 3;
if (++a < 4)
  if (++a < 4)
    System.out.println(a);
  else
    System.out.println(a);
```

nu va afişa nimic la execuţie întrucât ramura `else` este asociată ultimului `if`. În schimb, dacă adăugăm pentru delimitarea blocului primei instrucţiuni `if` două acolade, atunci obţinem codul:

```
int a = 3;
if (++a  < 4) {
  if (++a < 4)
    System.out.println(a);
}
else
  System.out.println(a);
```

care va afişa la execuţie valoarea `4`.

2.9.5. Instrucţiunea `switch`

Instrucţiunea `switch` transferă controlul uneia dintre mai multe instrucţiuni depinzând de valoarea unei expresii. Sintaxa generală este:

```
switch (<Expresie>) {
     default: <Instructiune0>
     case <ExpresieConstanta1> : <Instructiune1>
     case <ExpresieConstanta2> : <Instructiune2>
     . . .
     case <ExpresieConstantaN> : <InstructiuneN>
}
```

Ordinea în care apar clauzele `case` nu are importanţă, iar clauza `default` poate fi scrisă oriunde. Tipul lui `<Expresie>` trebuie să fie `char`, `byte`, `short` sau `int`, în caz contrar, apare o eroare de compilare. Corpul instrucţiunii `switch` trebuie să fie un bloc. Orice instrucţiune conţinută de bloc poate fi etichetată de una sau mai multe etichete `case` şi `default`. Aceste etichete se spune că sunt asociate instrucţiunii `switch`, ca valori ale expresiilor constante din etichetele `case`.

În cadrul unei instrucţiuni `switch`, următoarele cerinţe trebuie satisfăcute, altfel obţinem o eroare la compilare:

1. Fiecare valoare a lui `<ExpresieConstanta1>`, `<ExpresieConstanta2>`, ..., `<ExpresieConstantaN>` trebuie să fie asignabilă cu tipul lui `<Expresie>`;
2. Nu trebuie să existe două expresii constante care să aibă aceeaşi valoare;
3. Cel mult o etichetă `default` poate fi asociată cu aceeaşi instrucţiune `switch`.

Când este executată o instrucţiune `switch`, mai întâi se evaluează `<Expresie>`. Dacă evaluarea lui `<Expresie>` se termină anormal, atunci instrucţiunea `switch` se termină anormal. Altfel, execuţia continuă prin compararea valorii lui `<Expresie>` cu fiecare expresie `case`. Atunci:

1. Dacă una dintre constantele `case` (adică `<ExpresieConstantaK>`, unde `K` este `1`, `2`, ..., `N`) este egală cu valoarea lui `<Expresie>`, atunci se execută toate instrucţiunile după eticheta `case` până la întâlnirea primului `break`. Dacă toate aceste instrucţiuni se execută normal, atunci întreaga instrucţiune `switch` se termină normal.
2. Dacă nu există nici o etichetă care să se potrivească cu valoarea lui `<Expresie>` şi există o etichetă `default`, atunci se execută toate instrucţiunile după eticheta `default` până la întâlnirea primului `break`. Dacă toate aceste instrucţiuni se execută normal sau nu există nici o instrucţiune după eticheta `default`, atunci întreaga instrucţiune `switch` se termină normal.
3. Dacă nu există nici o etichetă `case` care să se potrivească cu valoarea lui `<Expresie>` şi nu există o etichetă `default`, atunci nu se execută nimic şi instrucţiunea `switch` se termină normal.

Dacă execuţia unei `<InstructiuniK>`, unde `K` este `1`, `2`, ..., `N`, se termină anormal din cauza unei instrucţiuni `break` fără etichetă, atunci nu se execută nimic şi instrucţiunea `switch` se termină normal. Dacă execuţia unei `<InstructiuniK>` se termină anormal din alt motiv, atunci instrucţiunea `switch` se termină anormal din acelaşi motiv. Cazul terminării anormale din cauza unei instrucţiuni `break` cu etichetă se rezolvă folosind regula generală pentru instrucţiuni etichetate.

Exemplul 2.9.3. Următorul program Java scoate în evidenţă folosirea instrucţiunii `switch`:

```
public class Switch {
  static void f(int k) {
    switch (k) {
      default:
        System.out.println("implicit");
        break;
      case 1:
        System.out.println("unu");
        break;
      case 2:
      case 3:
        System.out.println("doi sau trei");
        break;
      case 4:
      case 5:
        System.out.println("patru sau cinci");
    }
  }

  public static void main(String[] args) {
    f(1);
    f(2);
    f(3);
    f(4);
    f(6);
  }
}
```

La execuţie, acesta va afişa:

```
unu
doi sau trei
doi sau trei
patru sau cinci
implicit
```

Se observă posibilitatea utilizării mai multor expresii `case` asociate aceluiaşi bloc de instrucţiuni. Este esenţială şi terminarea unui caz cu ajutorul instrucţiunii `break`. Dacă, de exemplu, nu ar fi existat primul `break` asociat cazului `default`, atunci rezultatul ar fi fost:

```
unu
doi sau trei
doi sau trei
```

```
patru sau cinci
implicit
unu
```

2.9.6. Instrucţiunea `while`

Instrucţiunea `while` este întrebuinţată pentru execuţia repetată a unei instrucţiuni şi are sintaxa generală:

```
while (<Expresie>)  <Instructiune>
```

unde `<Expresie>` trebuie să aibă tipul `boolean`, altfel obţinem o eroare de compilare. Execuţia instrucţiunii `while` începe prin evaluarea lui `<Expresie>`. Dacă evaluarea lui `<Expresie>` se termină anormal, atunci şi instrucţiunea `while` se termină anormal. Altfel, execuţia continuă astfel:

1. Dacă valoarea lui `<Expresie>` este `true`, atunci se execută `<Instructiune>`.
   a) Dacă execuţia lui `<Instructiune>` se termină normal, atunci întreaga instrucţiune `while` se execută din nou, începând cu reevaluarea lui `<Expresie>`.
   b) Dacă execuţia lui `<Instructiune>` se termină anormal, atunci se execută pasul 3.
2. Dacă valoarea lui `<Expresie>` este `false`, atunci nu se execută nimic şi execuţia instrucţiunii `while` se termină normal. De aceea, dacă valoarea lui `<Expresie>` este `false` pentru prima evaluare (iteraţie), atunci `<Instructiune>` nu se execută.
3. Dacă execuţia lui `<Instructiune>` se termină anormal din cauza unui `break` fără etichete, atunci nu se execută nimic şi execuţia instrucţiunii `while` se termină normal.
4. Dacă execuţia lui `<Instructiune>` se termină anormal din cauza unui `continue` fără etichetă, atunci instrucţiunea `while` se execută încă o dată.
5. Dacă execuţia lui `<Instructiune>` se termină anormal din cauza unui `continue` cu eticheta E, atunci:
   a) Dacă instrucţiunea `while` are eticheta E, atunci întreaga instrucţiune `while` se execută încă o dată;
   b) Dacă instrucţiunea `while` nu are eticheta E, atunci întreaga instrucţiune `while` se termină anormal;
6. Dacă execuţia lui `<Instructiune>` se termină anormal din alt motiv, atunci întreaga instrucţiune `while` se termină anormal din acelaşi motiv. Cazul terminării anormale datorat unui `break` cu etichetă se tratează după regulile instrucţiunilor cu etichetă.

Astfel, o instrucţiune `while` se poate executa de zero ori (dacă `<Expresie>` este evaluată la `false` încă de la început), de un număr finit de ori (dacă `<Expresie>` se evaluează la `true` sau există un `break` în `<Instructiune>` (acesta este cazul cel mai des întâlnit) sau de o infinitate de ori (dacă `<Instructiune>` nu reuşeşte să implice evaluarea lui `<Expresie>` la `true` sau nu există un `break` în `<Instructiune>`. Acest ultim caz implică intrarea execuţiei programului într-o „buclă infinită". În general, se recomandă utilizarea lui `while` când nu se cunoaşte a priori numărul de iteraţii.

Exemplul 2.9.4. Iată un prim exemplu de utilizare a instrucţiunii `while` care se termină cu ajutorul instrucţiunii `break`:

```
public class WhileUnu {
  public static void main(String args[]) {
    int i = 0;
    while (true) {
      if (i == 1) break;
      i = (int)(Math.random() * 5);
      System.out.println("i = " + i);
    }
  }
}
```

Acesta va afişa (eventual) la execuţie:

```
i = 2
i = 3
 ...
i = 1
```

Execuţia programului de mai sus este nedeterministă, adică la două execuţii distincte este foarte posibil să obţinem rezultate diferite.

Exemplul 2.9.5. Un al doilea exemplu de utilizare a instrucţiunii `while` condiţionat (adică de ieşire cu satisfacerea condiţiei booleene, şi nu cu `break`):

```
public class WhileDoi {
  public static int f(int n) {
    return (int)(Math.random() * n);
  }

  public static void main(String args[]) {
    int i = 0;
    while (f(5) != 1) i++;
    System.out.println("Au fost " + i + " iteratii.");
  }
}
```

O posibilă execuţie a acestui program este:

```
Au fost 9 iteratii.
```

2.9.7. Instrucţiunea do

Ca şi `while`, instrucţiunea `do` face parte din categoria instrucţiunilor repetitive şi implică execuţia cel puţin o dată a unei instrucţiuni. Sintaxa generală este:

```
do
    <Instructiune>
while (<Expresie>);
```

Tipul lui `<Expresie>` trebuie să fie `boolean`, altfel se obţine eroare la compilare. Semantica instrucţiunii `do` este simplă. Astfel, se execută mai întâi `<Instructiune>`:

1. Dacă execuţia lui `<Instructiune>` se termină normal, atunci se evaluează `<Expresie>`. Dacă evaluarea lui `<Expresie>` se termină anormal, atunci şi instrucţiunea `do` se termină anormal. Altfel, avem:
   a) Dacă valoarea lui `<Expresie>` este `true`, se execută din nou instrucţiunea `do`;
   b) Dacă valoarea lui `<Expresie>` este `false`, nu se execută nimic şi instrucţiunea `do` se termină normal.
2. Dacă execuţia lui `<Instructiune>` se termină anormal, atunci:
   a) Dacă execuţia lui `<Instructiune>` se termină anormal din cauza unui `break` fără etichete, atunci nu se execută nimic şi instrucţiunea `do` se termină normal;
   b) Dacă execuţia lui `<Instructiune>` se termină anormal din cauza unui `continue` fără etichete, atunci se evaluează `<Expresie>` şi avem cazurile 1. a) şi respectiv 1. b);
   c) Dacă execuţia lui `<Instructiune>` se termină anormal din cauza unei etichete E, atunci avem:
      i. Dacă instrucţiunea `do` are eticheta E, atunci se evaluează `<Expresie>` şi avem cazurile 1. a) şi respectiv 1. b);
      ii. Dacă instrucţiunea `do` nu are eticheta E, atunci instrucţiunea `do` se termină anormal.
3. Dacă execuţia lui `<Instructiune>` se termină anormal din alt motiv, atunci instrucţiunea `do` se termină anormal din acelaşi motiv. Cazul terminării anormale datorat unui `break` cu etichetă se tratează după regulile instrucţiunilor cu etichetă.

Se pot reformula exemplele 2.9.4 şi 2.9.5 astfel încât să simuleze iteraţiile cu instrucţiunea `do`, şi nu cu `while`.

Exemplul 2.9.6. Programul citeşte un număr arbitrar de valori în virgulă flotantă de la tastatură, apoi afişează suma acestora. Programul conţine şi noţiuni de fişiere necesare citirii şirurilor de caractere de la tastatură.

```
import java.io.*;

public class DoWhile {
  public static void main(String[] args) {
    BufferedReader tastatura = new
       BufferedReader(new InputStreamReader(System.in), 1);
    double suma = 0.0, numarCitit = 0.0;
    String linie = "";
    do {
       try {
         System.out.flush();
         System.out.println(
          "Dati un numar in virgula flotanta");
         linie = tastatura.readLine();
         Double tempDouble = Double.valueOf(linie);
         numarCitit = tempDouble.doubleValue();
         System.out.println("numarCitit = " + numarCitit);
         suma += numarCitit;
         System.out.println("Doriti sa continuati (d/n)?");
         System.out.flush();
         linie = tastatura.readLine();
         System.out.println("Ati tastat " + linie);
       }
       catch (IOException e) {
         System.err.println("Intrare de la tastatura " +
           e.toString());
         System.exit(1);
       }
    }
    while (linie.equals("d") || linie.equals("D"));
    System.out.println("Suma numerelor citite este " + suma);
    try {
      tastatura.close();
    }
    catch(IOException e) {
     System.err.println("Eroare inchidere fisier de intrare "
        + e.toString());
     System.exit(2);
    }
  } // sfarsitul metodei main()
} // sfarsitul definitiei clasei DoWhile
```

Iată o posibilă execuţie a acestui program:

```
Dati un numar in virgula flotanta
2.98
numarCitit = 2.98
Doriti sa continuati (d/n)?
d
```

```
Ati tastat d
Dati un numar in virgula flotanta
4.78f
numarCitit = 4.78
Doriti sa continuati (d/n)?
d
Ati tastat d
Dati un numar in virgula flotanta
-3.12
numarCitit = -3.12
Doriti sa continuati (d/n)?
n
Ati tastat n
Suma numerelor citite este 4.64
```

2.9.8. Instrucţiunea `for`

Ultima instrucţiune din categoria celor repetitive este `for`. În general, aceasta este utilă când se cunoaşte numărul (maxim) de iteraţii care vor trebui efectuate.

Instrucţiunea `for` are două forme. Prima dintre ele are sintaxa generală:

```
for ([<ListaExpresieInitializare>] ; [<Expresie>] ;
[<ListaExpresieActualizare>])
    <Instructiune>
```

Cea de-a doua formă este:

```
for (<ParametruFormal> : <Expresie>)
<Instructiune>
```

Mai întâi vom prezenta prima formă, apoi vom reveni asupra celei de-a doua forme a instrucţiunii `for`.

După cum se poate observa, în prima formă a instrucţiunii `for` toate cele trei expresii sunt opţionale. `<ListaExpresieInitializare>` şi `<ListaExpresieActualizare>` reprezintă liste de expresii şi instrucţiuni separate prin virgulă. În plus, `<ListaExpresieInitializare>` poate conţine şi expresii de declarare a variabilelor locale. Tipul lui `<Expresie>` trebuie să fie `boolean`, altfel obţinem o eroare de compilare.

Iată cum se poate scrie instrucţiunea `for` de mai sus utilizând o instrucţiune `while` şi păstrând echivalenţa semantică:

```
{
  <ListaExpresieInitializare>;
  while (<Expresie>) {
    <Instructiune>
    <ListaExpresieActualizare>;
  }
}
```

Să vedem exact modul de execuţie a instrucţiunii `for` de mai sus începând cu `<ListaExpresieInitializare>`:

1. Dacă `<ListaExpresieInitializare>` este o listă de expresii instrucţiune, atunci expresiile sunt evaluate de la stânga la dreapta. Dacă vreo expresie se evaluează anormal, atunci întreaga instrucţiune `for` se termină anormal.
2. Dacă `<ListaExpresieInitializare>` este o declaraţie de variabilă locală, aceasta este executată ca şi cum ar fi fost o instrucţiune de declaraţie a unei variabile locale într-un bloc. În acest caz, domeniul variabilei locale declarate este în `<ListaExpresieInitializare>`, `<Expresie>`, `<ListaExpresieActualizare>` şi `<Instructiune>`. Dacă execuţia declaraţiei variabilei locale se termină anormal dintr-un anume motiv, atunci instrucţiunea `for` se termină anormal din acelaşi motiv.
3. Dacă `<ListaExpresieInitializare>` nu este prezentă, atunci nu se execută nimic.

O iteraţie a instrucţiunii `for` se execută astfel:

1. Dacă `<Expresie>` este prezentă, atunci aceasta se evaluează. Dacă `<Expresie>` se evaluează anormal, atunci întreaga instrucţiune `for` se termină anormal. Altfel:
   a) Dacă `<Expresie>` se evaluează la `true`, atunci se execută `<Instructiune>`.
   b) Dacă `<Expresie>` este evaluat la `false`, atunci nu se mai execută nimic şi instrucţiunea `for` se termină normal. Dacă `<Expresie>` se evaluează la `false` încă de la prima iteraţie, atunci nu se execută `<Instructiune>`.
2. Dacă `<Expresie>` nu este prezentă, atunci singura posibilitate de a termina execuţia instrucţiunii `for` este întâlnirea unei instrucţiuni `break`.

La execuţia lui `<Instructiune>` distingem următoarele cazuri:

1. Dacă execuţia lui `<Instructiune>` se termină anormal din cauza unui `break` fără etichetă, atunci nu se mai execută nimic, instrucţiunea `for` se termină normal.
2. Dacă execuţia lui `<Instructiune>` se termină anormal din cauza unui `continue` fără etichetă, atunci, în cazul în care există `<ListaExpresieActualizare>`, expresiile sunt evaluate de la stânga la dreapta, iar dacă vreo expresie se evaluează anormal, atunci instrucţiunea `for` se termină anormal. Dacă nu există `<ListaExpresieActualizare>`, atunci nu se execută nimic. Apoi, se trece la următoarea iteraţie a lui `for`.
3. Dacă execuţia lui `<Instructiune>` se termină anormal din cauza unui `continue` cu eticheta E, în cazul în care instrucţiunea `for` nu are eticheta E, atunci instrucţiunea `for` se va termina anormal, iar dacă are eticheta E, se continuă cu cele două etape de la pasul 5.
4. Dacă execuţia lui `<Instructiune>` se termină anormal din anumite motive, atunci instrucţiunea `for` se termină anormal din acelaşi motiv. Cazul terminării cu un `break` cu etichetă se face la fel ca la instrucţiunile etichetă.
5. Dacă execuţia lui `<Instructiune>` se termină normal, atunci se parcurg următoarele etape:

- dacă există `<ListaExpresieActualizare>`, atunci expresiile sunt evaluate de la stânga la dreapta (în cazul în care vreo expresie se evaluează anormal, atunci instrucţiunea `for` se termină anormal); dacă nu există `<ListaExpresieActualizare>`, atunci nu se execută nimic;
- se trece la următoarea iteraţie a instrucţiunii `for`.

Exemplul 2.9.7. Considerăm o secvenţă de cod Java pentru calculul sumei numerelor de la `1` la `n`. Presupunem că `i`, `n` şi `suma` sunt trei variabile de tip `int`. Variabila `n` are deja o valoare, de preferat mai mare decât `1`. O primă variantă ar fi:

```
for (suma = 0, i = 1; i <= n; ++i)
    suma += i;
```

Aceasta se poate scrie şi mai compact, astfel:

```
for (suma = 0, i = 1; i <= n; suma += i, ++i);
```

Atenţie însă la ordinea de scriere dintre `suma += i` şi `++i`. Inversând cele două instrucţiuni, se obţine un alt rezultat pentru `suma` (s-ar obţine valoarea `2+3+...+(n+1)` în loc de `1+2+...+n`).

Cum s-a menţionat şi în semantica instrucţiunii `for` de mai sus, cele trei părţi ale lui `for` pot lipsi (fiecare sau cumulat). Astfel, putem elimina `<ListaExpresieInitializare>` şi putem scrie un cod echivalent:

```
i = 1;
suma = 0;
for ( ; i <= n; ++i)
    suma += i;
```

Putem elimina şi `<ListaExpresieActualizare>`, obţinând un alt cod echivalent:

```
i = 1;
suma = 0;
for ( ; i <= n; )
    suma += i++;
```

Dacă, în schimb, lipseşte şi `<Expresie>`, trebuie să utilizăm instrucţiunea `break`, în caz contrar obţinem o buclă infinită.

```
i = 1;
suma = 0;
for ( ; ; ) {
    suma += i++;
    if (i > n) break;
}
```

Putem utiliza şi instrucţiunea `break` cu etichetă. Astfel, putem scrie codul echivalent:

```
i = 1;
suma = 0;
```

```
eticheta: for ( ; ; ) {
      suma += i++;
      if (i > n) break eticheta;
}
```

Exemplul 2.9.8. Iată şi un exemplu de utilizare a instrucţiunii `continue` în cadrul unei instrucţiuni `for`. Este vorba de calculul sumei numerelor întregi pozitive dintr-un tablou (cu `10` elemente).

```
public class SumaNumerelorPozitive {
   public static void main(String args[]) {
      int v[] = {3, 5, -3, 6, 2, -1, 8, 9, -4, 1};
      int suma = 0;
      for (int i = 0; i < 10; ++i) {
         if (v[i] < 0) continue;
         suma += v[i];
      }
      System.out.println("suma = " + suma);
   }
}
```

Ideea este simplă. Se parcurg toate elementele tabloului, iar pentru cele negative se sare peste instrucţiunea `suma += v[i];` (se reia `for`-ul de la început). Astfel se va afişa la execuţie `34`.

Să revenim la cea de-a doua formă a instrucţiunii `for`. `<ParametruFormal>` este de forma `<Tip> <numeVariabila>`, `<Tip>` fiind un tip Java, numit şi tipul parametrului formal. `<Expresie>` este o expresie care se evaluează la un tablou de elemente sau la un obiect a cărui clasă implementează interfaţa `Iterable`. În cazul tabloului, elementele sale trebuie să aibă acelaşi tip ca parametrul formal sau să se poată converti implicit la tipul acestuia. `<Instructiune>` are aceeaşi semnificaţie ca pentru prima formă.

Exemplul 2.9.9. Programul conţine parcurgerea a două tablouri, unul de tip `String`, iar celălalt de `int`-uri, folosind a doua formă a instrucţiunii `for`.

```
public class ForTablouri {
   public static void main(String[] args) {
      String[] fructe = {"portocale", "mere", "kiwi"};
      for (String fruct : fructe)
        System.out.println(fruct);

      int[] numere = {12, 0, -5, 2};
      for (long numar : numere)
        System.out.println(numar);
   }
}
```

La al doilea `for` se realizează conversia implicită de la tipul `int`, cel al elementelor tabloului, la tipul `long`, al parametrului formal `numar`.

În capitolul 4 sunt prezentate colecţiile din Java şi este utilizată forma a doua pentru `for`.

2.9.9. Instrucţiunea `break`

Instrucţiunea de transfer de control necondiţionat `break` are sintaxa generală:

```
break [<Identificator>];
```

unde `<Identificator>` este o etichetă opţională. O instrucţiune `break` fără etichetă transferă controlul celei mai interioare instrucţiuni `switch`, `while`, `do` sau `for` (care se mai numesc ţinte `break` şi care se termină normal). Astfel, o instrucţiune `break` fără etichetă întotdeauna se termină anormal, iar dacă nu există ţinte `break` (adică vreo instrucţiune `switch`, `while`, `do` sau `for`) atunci se obţine eroare de compilare.

O instrucţiune `break` cu eticheta E încearcă să transfere controlul instrucţiunii etichetate cu E (care se mai numeşte ţintă `break` şi care se termină normal) în care se află `break`. Ţinta `break` nu trebuie să fie neapărat o instrucţiune `switch`, `while`, `do` sau `for`. Astfel, o instrucţiune `break` cu etichetă întotdeauna se termină anormal, iar dacă nu există ţinte `break` cu eticheta E, atunci se obţine eroare de compilare.

Aşadar, chiar dacă instrucţiunea `break` are sau nu etichetă, aceasta se termină anormal.

Exemplele 2.6.3, 2.9.1, 2.9.3, 2.9.4, 2.9.7 conţin referiri la instrucţiunea `break`. Vom prezenta doar un exemplu simplu de folosire a lui `break` cu etichetă, dar în care ţinta `break` nu este o instrucţiune `switch`, `while`, `do` sau `for`.

Exemplul 2.9.10. Fie programul Java de mai jos:

```
public class BreakIf {
  public static void main(String args[]) {
    boolean b = false;
    eticheta: if ((int)(Math.random() * 2) == 0) {
      b = true;
      break eticheta;
    }
    if (b)
      System.out.println("S-a executat break eticheta.");
    else
      System.out.println("Nu s-a executat break eticheta.");
  }
}
```

La unele execuţii, programul de mai sus va afişa:

```
S-a executat break eticheta.
```

iar la altele:

```
Nu s-a executat break eticheta.
```

2.9.10. Instrucţiunea `continue`

Instrucţiunea `continue` poate apărea doar în interiorul unor instrucţiuni repetitive (`while`, `do`, `for`) şi are sintaxa generală:

```
continue [<Identificator>];
```

unde `<Identificator>` se numeşte eticheta instrucţiunii `continue`. O instrucţiune `continue` fără etichetă încearcă transferarea controlului celui mai interior `while`, `do`, `for`, care se numesc ţinte `continue`. Apoi termină imediat iteraţia curentă şi începe una nouă. O instrucţiune `continue` se termină anormal, iar dacă nu există instrucţiunile `while`, `do` sau `for`, atunci se obţine o eroare la compilare.

O instrucţiune `continue` cu eticheta E încearcă transferarea controlului instrucţiunii în care se află, dar care are eticheta E. Apoi termină imediat iteraţia curentă şi începe una nouă. O instrucţiune `continue` se termină anormal, iar dacă nu există o instrucţiune `while`, `do` sau `for` etichetată cu E în care se află instrucţiunea `continue`, atunci se obţine o eroare la compilare.

Aşadar, o instrucţiune `continue` se termină întotdeauna anormal. Exemplul 2.9.8 conţine referiri la instrucţiunea `continue` fără utilizarea etichetelor. Vom prezenta un exemplu simplu în care folosim `continue` cu etichete.

Exemplul 2.9.11. Această aplicaţie simplă pune în evidenţă suma elementelor de tip `double` ale unei matrice cu `3` linii şi `4` coloane. La calculul sumei participă doar elementele care sunt mai mari în modul decât `0.01`. Se va utiliza instrucţiunea `continue` care poate transfera necondiţionat controlul programului la eticheta `unu` sau `doi`.

```
public class Continue {
  public static void main(String args[]) {
    // m  numarul de linii, n - numarul de coloane
    int m = 3, n = 4;
    // i, j sunt variabile de lucru
    int i, j;
    // a este o matrice cu 3 linii si 4 coloane cu
    // elemente de tip double
    double [][] a = { {2.45, -5.67, 0.001, 5.32},
                      {5.65, 2.28, 4.01, 0.002},
                      {-0.005, 6.39, 8.13, -7.89}
                    };
    // suma este o variabila de tip double care tine
    // suma elementelor mai mari in modul decat 0.01
    double suma = 0;
```

```
      unu: for (i = 0; i < m; i++) {
        doi: for (j = 0; j < n; j++) {
          if (Math.abs(a[i][j]) < 0.01)
            if (j < n - 1) continue doi;
            else continue unu;
          suma += a[i][j];
        }
      }
      System.out.println(
        "Suma elementelor mai mari decat 0.01 este " + suma);
  }
}
```

Execuţia sa va afişa textul:

```
Suma elementelor mai mari decat 0.01 este 20.67
```

2.9.11. Instrucţiunea `return`

Instrucţiunea `return` cedează controlul apelului unei metode sau constructor şi are sintaxa generală:

```
return [<Expresie>];
```

O instrucţiune `return` fără expresie trebuie să se afle în corpul unei metode care returnează `void` sau al unui constructor. O instrucţiune `return` nu poate apărea într-un iniţializator static, deoarece acesta este executat la iniţializarea clasei şi nu se poate termina anormal. În schimb, instrucţiunea `return` încearcă transferul controlului către apelantul metodei sau constructorul acesteia. Mai precis, o instrucţiune `return` fără expresie se va termina anormal.

O instrucţiune `return` cu expresie trebuie să fie conţinută de o declaraţie de metodă care returnează o valoare, în caz contrar apare o eroare de compilare. `<Expresie>` trebuie să fie o expresie care se evaluează la o valoare de tip T, iar tipul T trebuie să fie asignabil tipului returnat de metodă, în caz contrar apare o eroare de compilare. La execuţia instrucţiunii `return <Expresie>` se evaluează mai întâi `<Expresie>`. Dacă aceasta eşuează dintr-un anume motiv, atunci instrucţiunea `return` se termină anormal din acelaşi motiv. Dacă evaluarea lui `<Expresie>` se termină normal şi produce valoarea V, atunci instrucţiunea `return` se termină anormal, returnând valoarea V.

Aşadar, instrucţiunea `return` se termină anormal întotdeauna.

Exemplele 2.4.8, 2.6.3, 2.8.8 şi 2.9.5 se referă la instrucţiunea `return`. Vom prezenta un exemplu referitor la iniţializatori statici.

Exemplul 2.9.12. Următorul program va furniza o eroare la compilare, deoarece avem o instrucţiune `return` într-un iniţializator static:

```
public class InitializatorStatic {
  static int i = 6;
  static {
    System.out.println("Cod static: i = " + i++);
    return i;
  }
  public static void main(String[] args) {
    System.out.println("Cod in main(): i = " + i++);
    return ;
  }
}
```

Evident, corectarea se poate realiza prin eliminarea instrucţiunii `return i;` din iniţializatorul static. După corecţie, programul va afişa la execuţie:

```
Cod static: i = 6
Cod in main(): i = 7
```

Observăm că instrucţiunea `return` se poate folosi fără probleme în metode statice, în acest caz, în metoda `main()`.

2.9.12. Instrucţiunea **`throw`**

Termenul *excepţie* din Java se referă la prevenirea apariţiei unei erori care are şanse mari de apariţie (sau altfel spus se referă la tratarea cazurilor particulare). De exemplu, dacă dorim să încărcăm un fişier care nu există pe disc, dacă un fişier conţine date corupte, dacă o conexiune de reţea eşuează, dacă există în program o încercare de accesare ilegală a memoriei, atunci trebuie să tratăm aceste cazuri. Excepţiile sunt indicate prin aruncarea (engl. *throw*) lor şi sunt detectate prin prinderea (engl. *catch*) acestora. Excepţiile sunt, de fapt, obiecte ale clasei `java.lang.Throwable`.

Instrucţiunea `throw` implică aruncarea unei excepţii şi are sintaxa generală:

```
throw <Expresie>;
```

Rezultatul este transferul imediat al controlului din care pot ieşi instrucţiunile, constructorii, evaluările de iniţializare a câmpurilor şi apelurile metodelor până când se găseşte o instrucţiune `try` care „prinde" (se unifică cu) valoarea aruncată. Dacă nu este găsită nici o instrucţiune `try`, atunci execuţia firului responsabil cu `throw` este terminată după apelul metodei `UncaughtException` pentru grupul de fire de execuţie căruia îi aparţine firul în cauză. `<Expresie>` trebuie să fie o variabilă sau o valoare a unui tip referinţă asignabil la tipul `Throwable`, în caz contrar apare o eroare la compilare. În plus, una dintre următoarele trei condiţii trebuie să aibă loc:

1. Dacă excepţia nu este excepţie verificată, atunci una dintre situaţii este adevărată:
   a) Tipul lui `<Expresie>` este clasa `RuntimeException` sau o subclasă a sa;
   b) Tipul lui `<Expresie>` este clasa `Error` sau o subclasă a sa.

2. Instrucţiunea `throw` este conţinută într-un bloc `try` (al unei instrucţiuni `try`) şi tipul lui `<Expresie>` este asignabil cu tipul parametrului a cel puţin unei clauze `catch` a instrucţiunii `try`.
3. Instrucţiunea `throw` este conţinută de o declaraţie de metodă sau de un constructor şi tipul lui `<Expresie>` este asignabil cel puţin unui tip listat în clauzele `throws`.

Instrucţiunea `throw` evaluează mai întâi `<Expresie>`. Dacă evaluarea lui `<Expresie>` se termină anormal dintr-un anume motiv, atunci instrucţiunea `throw` se termină anormal din acelaşi motiv. Dacă evaluarea lui `<Expresie>` se termină normal, producând valoarea V, atunci instrucţiunea `throw` se termină anormal, aruncându-se valoarea V. Instrucţiunea `throw` se termină anormal mereu.

Exemplul 2.9.13. Reluăm metoda `parseInt()` din clasa `Integer`, care apare în exemplele 2.5.18 şi 2.6.3. Pentru a vedea ce excepţie aruncă o metodă dintr-o bibliotecă Java ne uităm la prototipul ei. Codul metodei `parseInt()` arată cam aşa (doar structura):

```
public static int parseInt(String s) throws
NumberFormatException {
      ...
   /* codul metodei */
      ...
   throw new NumberFormatException();
      ... //etc.
}
```

Exemplul 2.9.14. Vom descrie o metodă care poate arunca excepţii utilizând instrucţiunea `throw`.

```
public class TestExceptii {
   public static void main(String args[]) {
      for (int i = 0; i <= 2; i++) {
         try {
            System.out.println("\nTest caz " + i);
            metoda1(i);
            System.out.println("Sfarsit caz " + i);
         }
         catch (Exception e) {
            System.err.println("Exceptie: " + e.getMessage());
         }
         finally {
            System.out.println("Executam finally!");
         }
      }
   }
   private static void metoda1(int i) throws Exception {
      System.out.println("Incepe metoda1:");
      if (i != 0) throw new Exception("din metoda1");
      System.out.println("Sfarsit metoda1.");
   }
} // sfarsitul definitiei clasei TestExceptii
```

La execuţie se va afişa:

```
Test caz 0
Incepe metoda1:
Sfarsit metoda1.
Sfarsit caz 0
Executam finally!

Test caz 1
Incepe metoda1:
Exceptie: din metoda1
Executam finally!

Test caz 2
Incepe metoda1:
Exceptie: din metoda1
Executam finally!
```

Se observă că în cazul când `i` are valoarea `0`, nu se aruncă nici o excepţie. În schimb, când `i` are valoarea `1` sau `2`, `metoda1()` nu se mai execută în totalitate, ci numai până la aruncarea excepţiei.

2.9.13. Instrucţiunea **synchronized**

Instrucţiunea `synchronized` va implica satisfacerea proprietăţii de excludere mutuală şi anume executarea unui bloc de către cel mult un fir de execuţie. După execuţie, se eliberează resursa respectivă. Sintaxa generală a instrucţiunii `synchronized` este:

```
synchronized (<Expresie>) <Bloc>
```

Tipul lui `<Expresie>` trebuie să fie referinţă, în caz contrar apare o eroare la compilare. Instrucţiunea `synchronized` evaluează întâi `<Expresie>`. Dacă evaluarea se termină anormal dintr-un anume motiv, atunci instrucţiunea `synchronized` se termină anormal din acelaşi motiv. Dacă valoarea lui `<Expresie>` este `null`, atunci se aruncă o excepţie `NullPointerException`. Altfel, firul de execuţie închide resursa obţinută în urma evaluării. Apoi se execută `<Bloc>`. Dacă execuţia lui `<Bloc>` se termină normal, atunci resursa este eliberată şi instrucţiunea `synchronized` se termină normal. Dacă execuţia lui `<Bloc>` se termină anormal dintr-un anume motiv, atunci resursa este eliberată, iar instrucţiunea `synchronized` se termină anormal din acelaşi motiv.

Închiderile resurselor realizate de instrucţiunea `synchronized` **sunt la fel ca în** cazul metodelor sincronizate (care utilizează modificatorul de acces `synchronized`). Un singur fir de execuţie poate închide o resursă de mai multe ori.

Exemplul 2.9.15. Vom pune în evidenţă un exemplu simplu de închidere a unei resurse (un obiect al clasei respective).

```
public class InstructiuneaSynchronized {
  public static void main(String[] args) {
    InstructiuneaSynchronized is = new
    InstructiuneaSynchronized();
    synchronized(is) {
      synchronized(is) {
        System.out.println(
          "Acum obiectul este inchis de doua ori!");
      }
      System.out.println(
        "Acum obiectul este inchis o data!");
    }
    System.out.println(
      "Acum obiectul nu mai este inchis!");
  }
}
```

La execuţie, acest program va afişa:

```
Acum obiectul este inchis de doua ori!
Acum obiectul este inchis o data!
Acum obiectul nu mai este inchis!
```

2.9.14. Instrucţiunea `try-catch-finally`

Cum am prezentat în secţiunea 2.9.11, excepţiile sunt indicate prin aruncarea lor, prin utilizarea instrucţiunii `throw` şi sunt detectate prin prinderea lor, prin intermediul instrucţiunii `try-catch-finally`. Sintaxa generală a instrucţiunii `try-catch-finally` este:

```
Try <BlocTry>
   [catch (<ParametruFormal1>) <BlocCatch1>]
   [catch (<ParametruFormal2>) <BlocCatch2>]
      . . .
   [catch (<ParametruFormalN>) <BlocCatchN>]
   [finally <BlocFinally>]
```

cu restricţia că trebuie să existe o clauză `catch` sau `finally` (adică nu pot lipsi amândouă). O instrucţiune `try` poate avea mai multe clauze `catch` (numite şi *rezolvitori de excepţii*). Forma generală a lui `<ParametruFormalK>`, unde `K` este `1`, `2`, ..., N, este `<TipK>` `<ParametruExceptieK>`, iar `<TipK>` trebuie să fie din clasa `Throwable` sau o subclasă a acesteia, în caz contrar apare o eroare la compilare. Domeniul variabilei parametru `<ParametruExceptieK>` este `<BlocCatchK>`. Numele lui `<ParametruExceptieK>` trebuie să difere de numele altor variabile locale sau parametri care au acelaşi domeniu, în caz contrar apare o eroare la compilare.

Dacă este aruncată o valoare şi instrucţiunea `try` are mai multe clauze `catch` care o pot prinde, atunci controlul se transferă primei clauze `catch` cu care se unifică. Din acest motiv, în Java, blocul `catch` mai specific trebuie să le preceadă pe cele mai generale, în caz contrar se obţine eroare la compilare. Clauza `finally` se execută întotdeauna, chiar dacă se prinde vreo excepţie sau nu.

Începem cu prezentarea lui `try-catch` (clauza `finally` va fi explicată ulterior). O instrucţiune `try` fără blocul `finally` începe prin execuţia lui `<BlocTry>`. Atunci:

1. Dacă execuţia lui `<BlocTry>` se termină normal, atunci nu se mai execută nimic în plus şi spunem că instrucţiunea `try` s-a terminat normal;
2. Dacă execuţia lui `<BlocTry>` nu se termină normal din cauza unei instrucţiuni `throw` de valoare V, atunci:
   a) Dacă tipul lui V este asignabil lui `<TipK>`, unde `K` este `1`, `2`, ..., `N`, atunci se selectează prima clauză `catch` pentru care V se unifică cu `<ParametruExceptieK>`, apoi se execută `<BlocCatchK>`. Dacă blocul `<BlocCatchK>` se termină normal, atunci instrucţiunea `try` se termină normal. Dacă blocul `<BlocCatchK>` se termină anormal dintr-un anume motiv, atunci instrucţiunea `try` se termină anormal din acelaşi motiv.
   b) Dacă tipul lui V nu este asignabil lui `<TipK>`, `K` este `1`, `2`, ..., `N`, atunci instrucţiunea `try` se termină anormal din cauza unei instrucţiuni `throw` de valoare V.
3. Dacă `<BlocTry>` se termină anormal dintr-un anume motiv, atunci instrucţiunea `try` se termină anormal din acelaşi motiv.

Exemplul 2.9.16. Vom prezenta un program Java care compară aruncarea propriilor excepţii faţă de excepţiile sistem (`RuntimeException`).

```
class ExceptiaNoastra extends Exception {
  ExceptiaNoastra() { }
  ExceptiaNoastra(String s) { super(s); }
}

public class Exceptie {
  static void f() throws ExceptiaNoastra {
    throw new ExceptiaNoastra();
  }

  public static void main(String[] args) {
    try {
      f();
    }
```

```
      catch (RuntimeException re) {
        System.err.println("S-a aruncat exceptia: " + re);
      }
      catch (ExceptiaNoastra en) {
        System.err.println("S-a aruncat exceptia: " + en);
      }
    }
  } // sfarsitul definitiei clasei Exceptie
```

La execuţie se va afişa:

```
S-a aruncat exceptia: ExceptiaNoastra
```

Exemplul 2.9.17. Următorul subprogram Java poate genera o împărţire (nepermisă) la `0` (va arunca o excepţie `ArithmeticException`) sau o depăşire de indice de vector (va arunca o excepţie `ArrayIndexOutOfBoundsException`).

```
int x = (int) (Math.random() * 5);
int y = (int) (Math.random() * 10);
int [] z = new int[5];
try {
  System.out.println("y / x = " + (y / x));
  System.out.println("y = " + y + " z[y] = " + z[y]);
}
catch (ArithmeticException e) {
  System.err.println("Problema aritmetica: " + e);
}
catch (ArrayIndexOutOfBoundsException e) {
  System.err.println("Depasire de domeniu index vector " + e);
}
```

La execuţie, dacă nu apare nici o excepţie, atunci subprogramul îşi desfăşoară fluxul normal. Dacă apare o excepţie necaptată, atunci execuţia subprogramului se opreşte. Dacă există o instrucţiune `try`, dar nu există clauză `catch` corespunzătoare cu excepţia apărută, se întrerupe execuţia subprogramului. Dacă însă se găseşte un `catch` corespunzător, atunci acesta se execută şi se continuă execuţia subprogramului cu instrucţiunea următoare construcţiei `try`.

Continuăm cu semantica instrucţiunii `try-catch-finally` (adică există blocul `finally`). O instrucţiune `try` cu bloc `finally` începe cu execuţia lui `<BlocTry>`. Atunci:

1. Dacă execuţia lui `<BlocTry>` se termină normal, atunci se execută `<BlocFinally>` şi avem:
    a) Dacă `<BlocFinally>` s-a terminat normal, atunci instrucţiunea `try` se termină normal;
    b) Dacă `<BlocFinally>` s-a terminat anormal dintr-un motiv, atunci instrucţiunea `try` se termină anormal din acelaşi motiv.

2. Dacă execuţia lui `<BlocTry>` nu se termină normal din cauza unei instrucţiuni `throw` de valoare V, atunci:
   a) Dacă tipul lui V este asignabil lui `<TipK>`, iar `K` este `1`, `2`, ..., `N`, atunci se selectează prima clauză `catch` pentru care V se unifică cu `<ParametruExceptieK>`, apoi se execută `<BlocCatchK>`.
      i. Dacă blocul `<BlocCatchK>` se termină normal, atunci se execută `<BlocFinally>` şi avem în continuare cazurile 1. a) sau 1. b).
      ii. Dacă blocul `<BlocCatchK>` se termină anormal dintr-un anume motiv M1, atunci se execută `<BlocFinally>` şi dacă se termină normal, atunci instrucţiunea `try` se termină anormal din motivul M1, iar dacă `<BlocFinally>` se termină anormal din motivul M2, atunci instrucţiunea `try` se termină anormal din motivul M2 (şi nu din motivul M1).
   b) Dacă tipul lui V nu este asignabil lui `<TipK>`, `K` este `1`, `2`, ..., `N`, atunci se execută `<BlocFinally>` şi distingem situaţiile:
      i. Dacă `<BlocFinally>` se termină normal, atunci instrucţiunea `try` se termină anormal din cauza unei instrucţiuni `throw` de valoare V;
      ii. Dacă `<BlocFinally>` se termină anormal din motivul M1, atunci instrucţiunea `try` se termină anormal din motivul M1 (valoarea V este eliberată).
3. Dacă `<BlocTry>` se termină anormal din motivul M1, atunci se execută `<BlocFinally>` şi distingem situaţiile:
   a) Dacă `<BlocFinally>` se termină normal, atunci instrucţiunea `try` se termină anormal din motivul M1;
   b) Dacă `<BlocFinally>` se termină anormal din motivul M2, atunci instrucţiunea `try` se termină anormal din motivul M2.

Exemplul 2.9.18. Continuăm ideea din exemplul 2.9.17 de a defini propria noastră clasă de excepţii şi punem în evidenţă utilizarea lui `finally`:

```
class ExceptiaNoastra extends Exception {
  ExceptiaNoastra() { }
  ExceptiaNoastra(String s) { super(s); }
}

public class ExceptieFinally {
  static void f() throws ExceptiaNoastra {
    throw new  NullPointerException();
  }

  public static void main(String[] args) {
    try {
      f();
    }
    catch (ExceptiaNoastra re) {
        System.err.println("S-a aruncat exceptia: " + re);
      }
      finally {
        System.err.println("Exceptie neprinsa.");
      }
    }
  } // sfarsitul definitiei clasei ExceptieFinally
```

La execuţie, programul va afişa:

```
Exceptie neprinsa.
java.lang.NullPointerException
      at ExceptieFinally.f(ExceptieFinally.java:8)
      at ExceptieFinally.main(ExceptieFinally.java:13)
```

Clasa de excepţii `NullPointerException` (derivată din `RuntimeException`) care este aruncată de metoda `f()` nu este prinsă de instrucţiunea `try` din `main()`, deoarece un obiect `NullPointerException` nu este asignabil unei variabile de tip `ExceptiaNoastra`. Aceasta implică execuţia clauzei `finally` (care afişează prima linie), apoi firul de execuţie asociat lui `main()` va afişa mesajele de eroare ulterioare.

În principiu, există patru situaţii când nu se execută instrucţiunile din corpul `finally`: oprirea firului de execuţie curent, apelul `System.exit()`, oprirea calculatorului sau apariţia unei excepţii în blocul `finally`.

Exemplul 2.9.19. Să presupunem că un program deschide cu succes un fişier, dar apare o eroare în timpul citirii din fişier. Este de dorit închiderea fişierului în această situaţie. Pentru aceasta trebuie să utilizăm `finally`.

```
String linie;
try {
  while ((linie=fisier.readLine()) != null) {
    ...//procesare linie
  }
}
catch (IOException e) {
  campEroare.setText("Eroare in fisierul de intrare");
}
finally {
  fisier.close();
}
```

Astfel, fişierul curent va fi închis indiferent de excepţia apărută.

2.10. Concluzii

Capitolul 2 se referă la fundamentele limbajului Java, în special la „aspectul static”, tipul referinţă urmând să fie prezentat în capitolul 3.

Secţiunea 2 prezintă fişierele-sursă. Acestea trebuie să aibă extensia `.java` şi să conţină cel mult o definiţie a unei clase de bază publice, al cărei nume trebuie să coincidă cu numele fişierului (bineînţeles, fără extensie). Un fişier-sursă poate conţine un număr nelimitat de definiţii de clase nepublice.

Secţiunea 3 se referă la atomi lexicali. Prima etapă a compilării unui program Java începe cu *analizarea lexicală* a codului-sursă Java. Asta înseamnă parcurgerea secvenţială a textului identificând cuvinte acceptate de sintaxa Java, numite *atomi lexicali* (engl. *tokens*). Aceşti atomi lexicali sunt: identificatori, cuvinte rezervate, literali, separatori şi operatori ai limbajului Java. Spaţiile şi comentariile sunt, de asemenea, identificate, însă acestea nu mai sunt transmise copiei programului Java în vederea continuării procesului de compilare (analiză sintactică, semantică etc.). Înaintea prezentării traducerilor lexicale, se descriu caracterele în care sunt scrise programele Java, aşa-numitele *caractere Unicode*. Traducerea lexicală poate folosi orice secvenţă Unicode cu sens special (engl. *escape*) pentru a include orice caracter Unicode folosind doar caractere ASCII (engl. *American Standard Code for Information Interchange*).

Urmează secţiunea 4 referitoare la tipuri de date. Acestea trebuie cunoscute în momentul compilării pentru fiecare variabilă şi expresie, fapt pentru care se spune că Java este un limbaj *puternic tipizat*. În Java, există trei categorii de tipuri de date: *primitive*, *referinţă* şi `null`. Tipurile primitive sunt tipurile *numerice* şi `boolean`. Tipurile numerice sunt tipurile *integrale* (`byte`, `short`, `int`, `long` şi `char`) şi tipurile în *virgulă flotantă* (`float` şi `double`). Tipurile referinţă sunt tipurile *clasă*, *interfaţă* şi *tablou*. Un obiect în Java este o instanţă creată dinamic a unui tip clasă sau a unui tablou creat dinamic. Valorile unui tip referinţă sunt referinţe la obiecte. Toate obiectele, inclusiv tablourile, suportă metode ale clasei `Object`. Literalii `String` sunt reprezentaţi de obiecte `String`. Tipul `null` este special, acesta neavând nume.

Secţiunea 5 se referă la expresii şi operatori. Un aspect important în Java îl reprezintă *evaluarea expresiilor*, fie prin *efectele laterale* (asignarea de valori către variabile), fie pentru *valorile returnate* de acestea (utilizate ca argumente sau operanzi în alte expresii) sau pentru *afectarea secvenţei de execuţie* în instrucţiuni. Dacă o expresie este o variabilă sau o valoare, atunci expresia trebuie să aibă tipul cunoscut încă de la compilare. Ca şi la variabile, valoarea unei expresii este compatibilă cu tipul expresiei. Astfel, valoarea unei expresii al cărei tip este T poate fi asignată unei variabile de tip T. Dacă tipul unei expresii este un tip primitiv, atunci valoarea expresiei este de acelaşi tip primitiv. Dacă tipul expresiei este un tip referinţă, atunci clasa şi valoarea obiectului referenţiat nu se cunosc la momentul compilării, ci la al execuţiei. Prin cele 31 de exemple, secţiunea 5 prezintă expresiile şi operatorii referitori la tipurile primitive de date, cele de tip referinţă urmând a fi prezentate în capitolul „Clase, interfeţe şi tablouri”.

Secţiunea 6 prezintă noţiunea de *variabilă* în Java. Aceasta este o locaţie de memorie care are un tip asociat (tip verificat la compilare). O variabilă conţine o valoare care este compatibilă cu tipul variabilei. Valoarea unei variabile se poate schimba în urma unei asignări sau a folosirii operatorilor ++ şi --. Compatibilitatea valorilor implicite şi toate asignările unei variabile sunt verificate în timpul compilării, cu excepţia tabloului, când verificarea se realizează în timpul execuţiei programului Java. Exemplul 2.6.3 pune în evidenţă o mare parte a tipurilor de variabile, cele de tip instanţă şi de tip tablou urmând a fi prezentate în capitolul 3.

Secţiunea 7 se referă la *declaraţii* şi *iniţializări*. Fiecare instrucţiune de declarare a unei variabile locale este conţinută într-un bloc. Instrucţiunile de declarare a variabilelor locale pot fi intercalate printre alte instrucţiuni ale blocului. O declarare de variabilă locală poate apărea şi într-o instrucţiune `for`, caz în care se execută la fel ca o instrucţiune de declarare a unei variabile locale. O instrucţiune de declarare a unei variabile locale este o instrucţiune executabilă. La execuţie, declaratorii sunt procesaţi de la stânga la dreapta. Dacă un declarator are o expresie de iniţializare, atunci expresia este evaluată şi valoarea se asignează variabilei. Dacă declaratorul nu are o expresie de iniţializare, atunci compilatorul Java verifică dacă fiecare referinţă către variabilă este precedată de execuţia unei asignări a variabilei. Dacă nu este aşa, atunci se semnalează o eroare la compilare. Fiecare iniţializare (cu excepţia celei dintâi) este executată dacă precedenta expresie de iniţializare se evaluează normal. Execuţia unei declaraţii a unei variabile locale se termină normal dacă evaluarea ultimei expresii de iniţializare se face normal. Evident, dacă declaraţia unei variabile locale nu conţine expresii de iniţializare, atunci execuţia acesteia se termină întotdeauna normal.

Secţiunea 8 prezintă noţiunea de *conversie*. Aceasta are rolul de a modifica tipul unei expresii într-un anume context. În fiecare context sunt permise doar câteva conversii specifice: conversii identice, conversii primitive implicite (de la mic la mare, engl. *widening conversions*), conversii primitive explicite (de la mare la mic, engl. *narrowing conversions*), conversii de referinţă implicite, conversii de referinţă explicite, conversii la `String`. Cele 11 exemple prezentate în secţiunea 8 se referă la conversiile identice, la cele primitive (implicite şi explicite) şi la conversii la `String`. În capitolul 4 vor fi prezentate conversiile de referinţă.

Secţiunea 9 prezintă noţiunea de *structuri de control* (*instrucţiuni*). Acestea au rolul de a controla execuţia unui program Java prin efectul lor şi nu pentru faptul că întorc valori (cum era cazul expresiilor). Anumite instrucţiuni conţin alte instrucţiuni (sau expresii) ca parte a structurii lor. Instrucţiunile Java sunt: instrucţiunea vidă, etichetele, expresiile, `if`, `switch`, `while`, `do`, `for`, `break`, `continue`, `return`, `throw`, `synchronized`, `try`. Spre deosebire de alte limbaje de programare, în Java nu există instrucţiunea `goto`. Cele 19 exemple ale secţiunii pun în evidenţă modul de utilizare a acestor instrucţiuni.

2.11. Test-grilă

Întrebarea 2.11.1. Fie următoarea secvenţă de cod Java:

```
1) float a = 2;
2) float b = 1.3;
```

Care afirmaţie este adevărată?

a) Instrucţiunile 1) şi 2) sunt corecte.
b) Instrucţiunea 1) este corectă, instrucţiunea 2) este incorectă.
c) Instrucţiunile 1) şi 2) sunt incorecte.

Întrebarea 2.11.2. Fie următoarea declaraţie Java:

```
public private int h;
```

Care afirmaţie este adevărată?

a) Variabila `h` va fi accesată în mod `public`, deoarece se ia în considerare primul modificator de acces.
b) Variabila `h` va fi accesată în mod `private`, deoarece se ia în considerare ultimul modificator de acces.
c) Va genera eroare la compilare, deoarece o variabilă nu poate fi în acelaşi timp accesată `public` şi `private`.

Întrebarea 2.11.3. Fie următorul subprogram Java:

```
int x = 0;
if (Double.isInfinite(2 / x))
   System.out.println("infinit");
else
   System.out.println("2 / 0");
```

Ce puteţi spune despre acesta?

a) Va genera eroare la compilare din cauza împărţirii la zero.
b) Va genera eroare la execuţie din cauza împărţirii la zero (se aruncă o excepţie `ArithmeticException`).
c) Programul este corect şi va afişa `infinit`.
d) Programul este corect şi va afişa `NaN`.

Întrebarea 2.11.4. La ce valoare se va evalua următoarea expresie:

```
2f<3d?+0.0:5<7?0.3:21
```

a) Expresia conţine erori sintactice.
b) Unele conversii trebuie realizate explicit.
c) Se va evalua la `0.0`.
d) Se va evalua la `0.3`.

Întrebarea 2.11.5. Fie următorul subprogram Java:

```
String s1 = "anul" + 200 + 2, s2 = 200 + 2 + "anul";
System.out.println("s1 = " + s1 + ", s2 = " + s2);
```

Ce va afişa la execuţia sa?

a) `s1 = anul202, s2 = 202anul`
b) `s1 = anul2002, s2 = 202anul`
c) `s1 = anul202, s2 = 2002anul`
d) `s1 = anul2002, s2 = 2002anul`
e) Eroare la execuţie: este necesară o conversie explicită!

Întrebarea 2.11.6. Fie următorul subprogram Java:

```
int as = 3, bs = 2, cs = 4;
System.out.print(((as < bs++) & (cs++ < bs)) + " ");
System.out.println(as + " " + bs + " " + cs);
System.out.print(((as < bs++) && (cs++ < bs++)) + " ");
System.out.println(as + " " + bs + " " + cs);
```

Ce se poate spune despre acesta?

a) Eroare la compilare: nu se poate aduna o valoare booleană cu un `String`;
b) Subprogramul se compilează şi la execuţie afişează:

```
false 3 3 5
false 3 4 4
```

c) Subprogramul se compilează şi la execuţie afişează:

```
false 3 3 5
false 3 4 5
```

d) Subprogramul se compilează şi la execuţie afişează:

```
false 3 3 5
false 3 5 6
```

Întrebarea 2.11.7. Fie următorul program Java:

```
public class Asignare {
  public static void main(String args[]) {
    int a = 3;
    int b = (a = 2) * a;
    int c = b * (b = 5);
    System.out.println("a = " + a + ", b = " + b + ", c = " + c);
  }
}
```

Ce va afişa acesta la execuţie?

a) `a = 2, b = 4, c = 20`
b) `a = 2, b = 5, c = 20`
c) `a = 2, b = 5, c = 25`
d) `a = 3, b = 6, c = 30`

Întrebarea 2.11.8. Fie următorul subprogram Java:

```
System.out.println((1<2)?5:(3<4)) + " ");
```

Ce se poate spune despre acesta?

a) Eroare la compilare: nu se poate converti o valoare booleană la un `int`.
b) Subprogramul se compilează şi la execuţie afişează `5`.
c) Subprogramul se compilează şi la execuţie afişează `37`.
d) Subprogramul se compilează şi la execuţie afişează `true`.

Întrebarea 2.11.9. Fie următorul subprogram Java:

```
double d = 2.95;
int i = 4;
System.out.println(++d>i?d:i);
```

Ce se poate spune despre acesta?

a) Eroare la compilare: nu putem converti o valoare `double` la `int`.
b) Subprogramul se compilează şi la execuţie afişează `3.95`.
c) Subprogramul se compilează şi la execuţie afişează `4`.
d) Subprogramul se compilează şi la execuţie afişează `4.0`.

Întrebarea 2.11.10. Fie următorul apel Java:

```
System.out.println(1 < 2 < 3);
```

Ce se poate spune despre acesta?

a) Eroare la compilare.
b) Eroare la execuţie.
c) Apelul se compilează şi la execuţie afişează `true`.
d) Apelul se compilează şi la execuţie afişează `false`.

Întrebarea 2.11.11. Fie următorul program Java:

```
public class Apel {
  static int f(short a) { return a * 2; }
  public static void main(String[] args) {
    System.out.println(f(3));
  }
}
```

Ce se poate spune despre acesta?

a) Eroare la compilare: nu există o definiţie a metodei `f()` cu parametrul de tip `int`.
b) Eroare la compilare: metoda `f()` nu întoarce o valoare de tip `short`.
c) Programul se compilează şi la execuţie afişează `6`.

Întrebarea 2.11.12. Fie următorul subprogram Java:

```
byte b = -7 >>> 1;
System.out.println(b);
```

Ce se poate spune despre acesta?

a) Eroare la compilare.
b) Eroare la execuţie.
c) Subprogramul se compilează şi la execuţie afişează `-3`.
d) Subprogramul se compilează şi la execuţie afişează `-4`.

Întrebarea 2.11.13. Fie următorul subprogram Java:

```java
byte b = 7 >>> 1;
System.out.println(b);
```

Ce se poate spune despre acesta?

a) Eroare la compilare.
b) Eroare la execuţie.
c) Subprogramul se compilează şi la execuţie afişează `3`.
d) Subprogramul se compilează şi la execuţie afişează `-3`.

Întrebarea 2.11.14. Fie următorul program Java:

```java
public class Program {
  static int x = 6;
  public static void main(String[] args) {
    System.out.print("x = " + x);
    int x = (x = 3) * x;
    System.out.print(", x = " + x);
  }
}
```

Ce se poate spune despre acesta?

a) Eroare la compilare: variabila x este declarată de două ori.
b) Programul se compilează şi la execuţie afişează `x = 6, x = 3`.
c) Programul se compilează şi la execuţie afişează `x = 6, x = 9`.
d) Programul se compilează şi la execuţie afişează `x = 6, x = 18`.

Întrebarea 2.11.15. Fie următorul program Java:

```java
public class Program {
  public static void main(String[] args) {
    float min = Float.NEGATIVE_INFINITY;
    float max = Float.POSITIVE_INFINITY;
    System.out.println((int)(char)min + " " + (int)(char)max);
  }
}
```

Ce se poate spune despre acesta?

a) Eroare la compilare: conversii imposibile.
b) Programul se compilează şi la execuţie afişează `0 255`.
c) Programul se compilează şi la execuţie afişează `0 65535`.
d) Programul se compilează şi la execuţie afişează `0 -1`.

Întrebarea 2.11.16. Fie următorul subprogram Java:

```
int a = 3;
if (a++ < 4)
  if (++a < 4)
    System.out.println(a);
  else
    System.out.println(a);
```

Ce se poate spune despre acesta?

a) Eroare la compilare.
b) Subprogramul se compilează şi la execuţie afişează `4`.
c) Subprogramul se compilează şi la execuţie afişează `5`.
d) Subprogramul se compilează şi la execuţie nu afişează nimic.

Întrebarea 2.11.17. Fie următorul program Java:

```
public class Program {
  static void f(int k) {
    switch (k) {
      default: System.out.print("i "); break;
      case 1: System.out.print("1 "); break;
      case 2: case 3: System.out.print("23 "); break;
      case 4: case 5: System.out.print("45 ");
    }
  }
  public static void main(String[] args) {
    for (int i = 0; i < 6; i++)
      f(i);
  }
}
```

Ce se poate spune despre acesta?

a) Eroare la compilare.
b) Programul se compilează şi la execuţie afişează `i 1 23 23 45 45`.
c) Programul se compilează şi la execuţie afişează `i 1 23 45`.
d) Programul se compilează şi la execuţie afişează `i 1 23 23 45 45 i`.

Întrebarea 2.11.18. Fie următorul subprogram Java:

```
int i = 1, suma = 0;
for ( ; ; ) {
  suma += i++;
  if (i > 5) break;
}
System.out.print(suma);
```

Ce se poate spune despre acesta?

a) Eroare la compilare: lipsesc părţile componente ale lui `for`.
b) Subprogramul se compilează şi la execuţie afişează `0`.
c) Subprogramul se compilează şi la execuţie afişează `10`.
d) Subprogramul se compilează şi la execuţie afişează `15`.

Întrebarea 2.11.19. Fie următorul program Java:

```
public class Program {
  public static void main(String args[]) {
    int v[] = {2, 4, -2, 8, -2}, i = 0, suma = 0;
    for (i = 0; i < 10; ++i) {
      if (v[i] < 0) continue;
      suma += v[i];
    }
    System.out.println("suma = " + suma);
  }
}
```

Ce se poate spune despre acesta?
a) Eroare la compilare.
b) Programul se compilează şi la execuţie afişează `0`.
c) Programul se compilează şi la execuţie afişează `14`.
d) Programul se compilează şi la execuţie afişează `-4`.

Întrebarea 2.11.20. Fie următorul program Java:

```
public class Subiecte {
  public static void main(String args[]) {
    float f = 4.50;
    System.out.println(f);
  }
}
```

Ce se poate spune despre acesta?
a) Eroare la compilare: nu se poate converti implicit `4.50` la `float`.
b) Programul se compilează şi la execuţie afişează `4.5`.
c) Programul se compilează şi la execuţie afişează `4.50`.
d) Programul se compilează şi la execuţie afişează `4.5000000`.

Întrebarea 2.11.21. Ce se poate spune despre programul de mai jos?

```
public class Test21 {
  public static void main(String[] args) {
    float[] numere = {12.3f, 5, -123, 0.55F};
    double suma = 0;
    for (double d : numere)
      suma += d;
  }
}
```

a) Se obţine eroare la compilare: nu se pot converti implicit valorile `5` şi `-123` la `float`.
b) Se obţine eroare la compilare întrucât instrucţiunea `for` este folosită incorect.
c) Programul este corect şi afişează suma elementelor tabloului `numere`.
d) Se obţine eroare la compilare: nu se pot converti implicit valorile tabloului de tip `float` la tipul `double` al parametrului formal `d`.

2.12. Exerciţii propuse spre implementare

Exerciţiul 2.12.1. Scrieţi un program Java care să convertească mile terestre în kilometri (şi invers). Formula de transformare este 1 milă = 1,6 km.

Exerciţiul 2.12.2. Scrieţi un program Java care să împacheteze o dată calendaristică într-o variabilă de tip `int`. Furnizaţi şi o funcţie de despachetare. De exemplu, 20 februarie 2007 se reprezintă ca un întreg astfel: primii 5 biţi sunt rezervaţi pentru zi, următorii 4 biţi pentru lună şi ultimii pentru an.

Exerciţiul 2.12.3. Scrieţi o clasă Java care generează numere pseudoaleatoare din domeniul `0...65535`. Presupunând că avem o valoare iniţială (sământă, engl. *seed*) r_0, pentru generarea următorului număr aleator din domeniul `0...65535` utilizăm formula $r_{Nou} = ((r_{Vechi}*25173)+13849)\%65536$.

Exerciţiul 2.12.4. Scrieţi un program Java care primeşte la intrare un număr de secunde şi întoarce numărul maxim de ore, de minute, de secunde care este echivalent ca timp cu numărul iniţial de secunde. De exemplu, 7384 secunde este echivalent cu 2 ore, 3 minute şi 4 secunde.

Exerciţiul 2.12.5. Folosind o buclă `while` şi fără a utiliza tablouri, scrieţi un program Java care calculează termenul al `n`-lea din şirul lui Fibonnacci. Şirul lui Fibonnacci este dat de recurenţa liniară de ordin 2:

$$a_1 = a_2 = 1;$$
$$a_{n+2} = a_{n+1} + a_n, \quad n >= 1$$

Implementaţi şi o generalizare la recurenţe liniare de ordin k, k != 1, cu coeficienţi arbitrari:

$$a_1 = a_2 = \dots = a_k = 1;$$
$$a_{n+k} = a_{n+k} + a_{n+k-1} + \dots + a_n, \quad n >= 1$$

Exerciţiul 2.12.6. Scrieţi un program Java care să calculeze cel mai mare divizor comun dintre a şi b (notat cu (a, b)), unde a, b sunt numere întregi, folosind algoritmul lui Euclid. De exemplu, (12, 8) = 4, deoarece se pot aplica împărţiri succesive, rezultând astfel: 12 = 8 * 1 + 4, 8 = 4 * 2 + 0.

Exercițiul 2.12.7. Să se scrie un program Java care să calculeze n! (n! = 1 * 2 * ... * n), unde 0 < n < 13 este un număr natural. Determinați n! și pentru n >= 13.

Exercițiul 2.12.8. Folosind structura `for`, scrieți un program Java care calculează următoarele formule logice (sub forma unei tabele de adevăr): b1 | b3 | b5 și b1 & b2 | b4 & b5, unde b1, b2, b3, b4, b5 sunt variabile logice (cu valori `false`, `true`).

Exercițiul 2.12.9. Să se scrie o metodă Java care să returneze minimul a trei numere de același tip primitiv (folosind operatorul condițional `? :` sau instrucțiunea `if`). Folosind această metodă, scrieți o metodă Java care să returneze minimul a patru numere.

Exercițiul 2.12.10. Fie funcția lui Collatz: f(n) = n/2, dacă n este par, și f(n) = 3 * n + 1, dacă n este impar. Să se scrie un program Java care determină k natural minim astfel încât f (f (... f(n) ...)) = 1, unde parantezele (,) apar de k ori. De exemplu, pentru n = 6 avem: f(6) = 3, f(3) = 10, f(10) = 5, f(5) = 16, f(16) = 8, f(8) = 4, f(4) = 2, f(2) =1, deci k natural minim este 8.

Exercițiul 2.12.11. Scrieți un program Java care calculează suma divizorilor naturali ai unui număr natural n. Un număr este *perfect* dacă este egal cu suma divizorilor proprii pozitivi (de exemplu: 28 = 1 + 2 + 4 + 7 + 14). Să se genereze primele patru numere perfecte.

Exercițiul 2.12.12. Presupunem că depunem o sumă (depozit la termen) într-o bancă ce oferă o dobândă de 5% (de exemplu) pe an. Să se calculeze suma finală după un anumit număr de ani (se va face capitalizarea contului, adică se va ține cont de „dobânda la dobândă").

Exercițiul 2.12.13. Folosind operatori pe biți, scrieți metode Java care:

a) testează dacă un număr de tip `short`/`int`/`long` este divizibil cu 2, 4, 8. Generalizare pentru divizibilitate cu 2^n;
b) testează dacă un număr este pozitiv sau negativ;
c) calculează pentru un n dat multiplii de 2, 4, ... Generalizare;
d) calculează pentru un n dat [n/2], [n/4], ... Generalizare.

2.13. Proiecte propuse spre implementare

Proiectul 2.13.1. (Jocul *cap-pajură*, folosind o simulare Monte-Carlo) Presupunem că dispunem de o monedă ideală (nemăsluită). Doi jucători aruncă cu moneda după următoarele reguli:

a) Se fac n aruncări în total.
b) Primul jucător aruncă moneda și celălalt spune *cap* sau *pajură*.
c) Dacă acesta *ghicește* care față a monedei va pica, atunci se inversează jucătorii (adică aruncă al doilea și primul încearcă să ghicească).
d) La sfârșit trebuie afișat scorul (și procentul de câștig al fiecăruia).

Proiectul 2.13.2. (*Conjectura lui Goldbach*) Orice număr par mai mare decât 2 se poate scrie ca sumă a două numere prime. Scrieţi un program Java care verifică această conjectură pentru numere situate între m şi n. De exemplu, dacă m = 700 şi n = 1100, atunci trebuie afişat:

```
700 = 17 + 683
702 = 11 + 691
704 =  3 + 701
         ...
1098 =  5 + 1093
1100 =  3 + 1097
```

Generalizare: scrieţi toate combinaţiile posibile de adunare a două numere prime egale cu un număr dat.

Proiectul 2.13.3. (Jocul *hârtie, pumn, foarfece*) Presupunem că avem doi jucători care folosesc mâna dreaptă pentru reprezentarea a trei obiecte: *hârtie* înseamnă palma întinsă, *pumn* înseamnă mâna strânsă sub formă de pumn şi *foarfece* înseamnă două degete depărtate (semnul victoriei). Ei îşi arată simultan mâna dreaptă în una din aceste configuraţii (de mai multe ori). Dacă ei arată acelaşi lucru, este remiză (nu câştigă nimeni). Dacă nu, se aplică una dintre următoarele trei reguli:

- Hârtia acoperă pumnul (deci palma întinsă câştigă faţă de pumn).
- Pumnul sparge foarfecele.
- Foarfecele taie hârtia.

Să se simuleze acest joc făcând un număr arbitrar de evenimente şi precizând scorul final. Se cere să se joace persoană-calculator şi, varianta a doua, calculator-calculator.

Proiectul 2.13.4. O *ruletă* este o maşină care selectează la întâmplare un număr între 0 şi 35. Jucătorul poate paria pe un număr par/impar sau poate paria pe un număr oarecare. Câştigul unui pariu par/impar se premiază cu 2/1 euro, cu excepţia cazurilor în care ruleta alege 0. Dacă jucătorul ghiceşte numărul selectat de ruletă, atunci este plătit cu 35/1 euro. Să se simuleze acest joc, ştiind că jucăm pariuri de 1 euro şi avem o sumă iniţială de *n* euro.

3. Clase, interfeţe şi tablouri

3.1. Cuvinte-cheie

- clase, atribute, metode, constructori, modificatori de acces
- enumerări, clase generice
- interfeţe, metode abstracte, clase abstracte
- tablouri
- excepţii, erori
- conversii ale tipului referinţă

3.2. Clase

Clasa este entitatea de bază a unui limbaj orientat pe obiecte. În Java este obligatorie definirea a cel puţin unei clase. Nu este însă obligatorie crearea unei instanţe (unui obiect) a acelei clase, caz în care se vor utiliza date şi metode statice.

O declaraţie de clasă în limbajul Java are sintaxa generală:

```
[<ModificatoriClasa>] class <Identificator> [
<ListaTipuriReferinta>]
 [extends <NumeClasa>]
 [implements <ListaInterfete>] {
   [<DeclaratiiCorpClasa>]
}
```

unde:

- `<ModificatoriClasa>` pot fi: `public`, `abstract`, `final` (cel mult o apariţie şi, de preferat, în această ordine);
- `<Identificator>` reprezintă numele clasei;
- `<ListaTipuriReferinta>` este o listă de tipuri referinţă, numite şi tipuri generice, de forma: `<TipReferinta1>, <TipReferinta2>, ..., <TipReferintaN>` şi este cuprinsă între simbolurile „<”şi „>”.
- `<NumeClasa>` este numele unei alte clase (denumită clasă de bază sau superclasă);
- `<ListaInterfete>` este o listă de nume de interfeţe de forma: `<NumeInterfata1>, <NumeInterfata2>, ..., <NumeInterfataM>`;
- `<DeclaratiiCorpClasa>` este o listă de declaraţii de forma:
  - `<DeclaratiiMembruClasa>`, care, la rândul ei, poate fi `<DeclaratieAtribut>` sau `<DeclaratieMetoda>`;
  - `<InitializatorStatic>`;
  - `<DeclaratiiConstructor>`.

Dacă o clasă este declarată într-un pachet numit `Pachet`, atunci numele complet (calificat) al clasei este `Pachet.<Identificator>` (se poate consulta exemplul 2.2.1). În caz contrar, dacă definim clasa într-un pachet fără nume, atunci numele complet al clasei este `<Identificator>` (vezi exemplele 2.4.8, 2.4.9 etc.).

Un constructor al unei clase este o metodă care are numele respectivei clase. Constructorii nu returnează nimic şi nu au tip (nici măcar `void`). Constructorii sunt apelaţi automat în momentul creării unui obiect din clasa respectivă şi sunt utilizaţi pentru iniţializarea datelor membre.

Dacă pentru o clasă nu declarăm nici un constructor, atunci la compilare se creează automat un constructor implicit care este public, nu are nici un parametru şi execuţia sa nu are nici un efect. A nu se confunda constructorul implicit (cel generat automat) cu cel fără nici un parametru (scris de programator). Dacă declarăm un constructor, atunci nu se mai generează constructorul implicit.

Destructorul este metoda care se apelează automat la distrugerea unui obiect. Destructorii sunt extrem de utilizaţi în limbajul de programare C++, mai ales pentru eliberarea resurselor ocupate de respectivul obiect. În Java, lucrurile stau altfel datorită colectorului de gunoaie, care are drept sarcină eliberarea automată a memoriei la care aplicaţia nu mai are nici o referinţă. Prin suprascrierea metodei `finalize()` din clasa `Object`, programatorul poate specifica o mulţime de acţiuni, care vor fi executate odată cu distrugerea obiectului.

Fie clasa `Cerc`:

```
class Cerc {
  // date membre
  int raza, x, y;
  // constructor fara parametri
  Cerc() {
    raza = 1;
    x = 20;
    y = 20;
  }
  // constructor cu parametri
  Cerc(int r, int a, int b) {
    raza = r;
    x = a;
    y = b;
  }
  // metoda publica
  public void setRaza(int r) { raza = r; }
} // terminarea clasei Cerc
```

Aceasta conţine trei date membre de tip `int` - `raza`, `x` şi `y` -, doi constructori, unul cu parametri şi unul fără parametri, şi o metodă publică `setRaza()`.

3.2.1. Domeniul de vizibilitate al numelui unei clase

Domeniul de vizibilitate al numelui unei clase este întreg pachetul din care aceasta face parte.

Exemplul 3.2.1. Programul de mai jos, intitulat `DV.java`, face parte din pachetul cu numele `Pachet`. Constructorul clasei `DV` va crea două obiecte din clasele `C1`, respectiv `C2`. Spre deosebire de exemplele precedente, clasa `C1` conţine un obiect din clasa `C2` şi invers. Deoarece domeniul de vizibilitate a unei clase este pachetul în care se află, rezultă că putem referi un obiect din clasa `C1` în clasa `C2` (şi reciproc). Dacă nu ar fi fost conţinute în acelaşi pachet, am fi obţinut eroare la compilare.

```
package Pachet;

public class DV {
  public DV(int k) { // constructor
    C2 cDoi = new C2(k);
    System.out.println(cDoi);
    C1 cUnu = new C1(cDoi);
    System.out.println(cUnu);
  }
}

class C1 {
  /* date membre */
  int x;
  C2 c2;
  /* constructor */
  C1(C2 c2) {
    this.x = c2.x;
    this.c2 = c2;
  }
  /* definirea unei metode */
  public String toString() {
    return "x = " + x + " c2 = " + c2;
  }
}

class C2 {
  /* date membre*/
  int x;
  C1 c1;
  /* constructor */
  C2(int x) {
    this.x = ++ x;
    this.c1 = new C1(this);
  }
  /* definirea unei metode */
  public String toString() {
    return "x = " + x;
  }
}
```

De exemplu, putem scrie un program de tip aplicaţie de sine stătătoare care va apela clasa `DV` (declarată `public`, deci accesibilă):

```
import Pachet.*;

public class DomeniuVizibilitate {
  public static void main(String args[]) {
    DV obiect = new DV(4);
    }
}
```

În urma execuţiei programului de mai sus se obţine:

```
x = 5
x = 5 c2 = x = 5
```

3.2.2. Modificatorii unei clase

În definiţia clasei, modificatorii unei clase, care sunt notaţi cu `<ModificatoriClasa>`, pot fi: `public`, `abstract`, `final` (de preferat, în această ordine). Aceştia pot avea cel mult o apariţie, în caz contrar apare o eroare la compilare.

Dacă o clasă este declarată `public`, atunci aceasta poate fi accesată de orice cod Java care poate accesa pachetul în care este definită această clasă. Dacă o clasă nu este declarată `public`, atunci aceasta poate fi accesată doar din pachetul în care este declarată.

O clasă este *abstractă* dacă aceasta este incomplet definită (conţine o metodă neimplementată). Doar clasele abstracte pot avea metode abstracte (neimplementate). Dacă o clasă conţine o metodă abstractă şi nu este declarată abstract, atunci se obţine o eroare la compilare.

O clasă C are metode abstracte în unul dintre cazurile:

1. Clasa C conţine o declaraţie a unei metode abstracte;
2. Clasa C moşteneşte o metodă abstractă de la o altă clasă;
3. Când există o interfaţă care implementează clasa C şi care declară sau moşteneşte o metodă (care este clar că este abstractă, deoarece o interfaţă nu are metodele definite) şi clasa C nu declară şi nici nu moşteneşte o metodă care o implementează.

Exemplul 3.2.2. În continuare prezentăm un program Java care defineşte o clasă derivată dintr-o clasă abstractă şi o interfaţă.

```
// definirea unei clase abstracte
abstract class C1 {
  // declararea datelor membre
  int x = 1, y = 2;
  // declararea unei metode
  void f(int x) {
    this.x += x;
    g(x);
  }
  // declararea unei metode abstracte
  abstract void g(int y);
}

// definirea unei interfete
```

```
interface I1 {
  public void h();
}

/* definirea unei metode care deriveaza clasa C1
   si implementează interfata I1 */
class C2 extends C1 implements I1 {
  int z = 10;
  // implementarea metodei abstracte din clasa C1
  void g(int y) {
    this.y += y;
  }
  // implementarea metodei mostenita din interfata I1
  public void h() {
   System.out.println("x = " + x + ", y = " + y + ", z = " + z);
  }
}

// definirea clasei care va utiliza un obiect de tip C2
public class Testare {
  public static void main(String args[]) {
    C2 obiect = new C2();
    obiect.f(4);
    obiect.g(5);
    obiect.h();
  }
}
```

Precizăm faptul că o interfaţă grupează un set de metode abstracte. Aceste metode sunt implicit abstracte, deci nu trebuie să mai fie precedate de cuvântul-cheie `abstract`.

Clasa `C1` trebuie declarată `abstract`, deoarece conţine metoda `g()` neimplementată. Aceasta este definită în clasa derivată `C2`. Interfaţa `I1` conţine metoda abstractă `h()` al cărei cod va fi definit în clasa `C2` care implementează `I1` (în caz contrar, ar fi trebuit să definim clasa `C2` abstractă). În clasa de aplicaţie `Testare`, dacă am fi încercat crearea unui obiect din clasa `C1` sau din interfaţa `I1`, am fi obţinut eroare la compilare.

În principiu, un tip clasă se declară `abstract` doar dacă se intenţionează definirea altor subclase care să definească restul metodelor nedefinite. Dacă însă intenţia este să prevenim instanţierea clasei, putem declara un singur constructor ca fiind `private` (şi nu `public`).

De obicei, clasele abstracte sunt utilizate pentru a reprezenta concepte abstracte, de exemplu, o componentă grafică. Apoi se derivează respectiva clasă pentru a reprezenta elemente concrete, de exemplu, un buton.

Exemplul 3.2.3. Programul Java de mai jos:

```
class C {
  int x = 10;
  private C() { x = 5; }
}

public class TestareInstantiere {
  public static void main(String args[]) {
    C obiect = new C(); // eroare la compilare
  }
}
```

implică eroare la compilare, deoarece se încearcă instanţierea unei clase care are un constructor `private`.

O clasă care are doar un constructor şi acesta este declarat `private` nu poate fi nici derivată. În acest caz, clasa derivată nu poate apela constructorul clasei de bază, deoarece el este `private`.

O altă modalitate de a preveni derivarea claselor este prin utilizarea modificatorului `final`. O clasă poate fi declarată `final`, dacă definiţia sa este completă şi nu sunt necesare definiri de subclase. Încercarea de definire a unei clase derivate dintr-o clasă declarată `final` conduce la o eroare la compilare. În plus, o clasă definită `final` şi `abstract` conduce, de asemenea, la o eroare de compilare. Deoarece o clasă declarată `final` nu poate avea subclase, metodele acesteia nu pot fi niciodată definite.

Exemplul 3.2.4. Iată cum se poate defini o clasă `final`:

```
public final class C extends Object {
}
```

Astfel, clasa `C` nu va putea fi moştenită, dar poate fi instanţiată, după cum se poate vedea în programul:

```
final class C {
  int x = 5;
}

public class ClasaFinal {
  public static void main(String args[]) {
    C obiect = new C();
    System.out.println(obiect.x);
  }
}
```

care va afişa la execuţie `5`.

3.2.3. Clase derivate

Clauza opţională `extends` din declaraţia unei clase (vezi sintaxa generală în secţiunea 3.2) precizează care este clasa imediat următoare clasei curente (este vorba de `<NumeClasa>`, care se mai numeşte *superclasă*). Mai precis, în declaraţia `C1 extends C2`, spunem că `C2` este *superclasa directă* (se mai numeşte *clasa de bază*) a lui `C1`, iar `C1` este *subclasa directă* (se mai numeşte *clasa derivată*) a lui `C2`. Singura clasă care nu are superclase este `Object` (din pachetul `java.lang`). Clasa `Object` este superclasa tuturor claselor Java. Dacă o declaraţie de clasă nu foloseşte explicit clauza `extends`, atunci clasa `Object` este superclasa directă. Clasa `C2` trebuie să fie accesibilă, altfel se obţine eroare la compilare (`C1` este în acelaşi pachet cu `C2` sau `C2` este declarată `public` într-un pachet accesibil). La fel, dacă `C2` este declarată `final`, atunci se obţine o eroare la compilare (clasele declarate final nu pot avea subclase).

Se observă în Java că o clasă se poate deriva din cel mult o altă clasă, spre deosebire de alte limbaje de programare orientate obiect (cum ar fi C++) care permit *derivarea multiplă* (adică o clasă se poate deriva din mai multe clase). Spre deosebire de limbajul C++, unde relaţia de derivare se poate asocia cu un graf, în Java, relaţia de derivare se poate asocia unui arbore cu rădăcina etichetată cu `Object`.

Relaţia de *derivare a claselor* (indirectă) este închiderea tranzitivă a relaţiei de derivare directă. Spunem că C_1 este derivată din C_N, dacă există clasele C_2, C_3, ..., C_{N-1} astfel încât C_1 este derivată direct din C_2, C_2 este derivată direct din C_3, ..., C_{N-1} este derivată direct din C_N. Clasele C_1, C_2, ..., C_N sunt distincte, în caz contrar se obţine eroare la compilare. Evident, derivarea directă este un caz particular de derivare indirectă (când `N=1`).

Exemplul 3.2.5. În programul Java de mai jos:

```
class C1 {
  int x;
}

class C2 extends C1 {
  int y;
}
```

avem următoarele relaţii de derivare directă:

a) clasa `C1` este clasă derivată direct din `Object` şi clasa `Object` este superclasa directă a lui `C1`;
b) clasa `C2` este clasă derivată direct din `C1` şi clasa `C1` este superclasa directă a lui `C2`;
c) din a) şi b) rezultă că `C2` este derivată (indirect) din clasa `Object`.

Exemplul 3.2.6. Dacă în definiţia unei clase derivate apare clasa însăşi, atunci se formează un circuit şi se obţine eroare la compilare, cum este cazul programului Java de mai jos, când clasa `C1` este derivată direct din clasa `C2` şi clasa `C2` este derivată direct din clasa `C1`. Astfel, `C1` este derivată (indirect) din `C1`, aspect care conduce la eroare la compilare.

```
class C1 extends C2 {
   int x;
}

class C2 extends C1 {
   int y;
}
```

3.2.4. Clasa `Object`

În Java, clasa `Object` este superclasă pentru toate clasele. Orice clasă din Java moşteneşte metodele din clasa `Object`. Cele mai importante sunt:

Prototipul metodei	Descrierea metodei
`protected Object` **`clone`**`()`	Creează şi returnează o copie a obiectului curent.
`public boolean` **`equals`**`(Object obj)`	Testează dacă obiectul specificat este echivalent (egal) cu cel curent.
`protected void` **`finalize`**`()`	Metoda este apelată de către *garbage collector* când acesta constată că nu mai există referinţe la obiectul curent.
`public Class<?>` **`getClass`**`()`	Returnează clasa din care face parte obiectul curent în momentul execuţiei.
`public int` **`hashCode`**`()`	Returnează codul asociat obiectului curent.
`public String` **`toString`**`()`	Returnează reprezentarea obiectului curent într-un `String`.

Metodele `notify()`, `notifyAll()` şi `wait()` sunt prezentate în capitolul destinat firelor de execuţie. Clasa `Object` posedă doar un constructor fără parametri.

Exemplul 3.2.7. Utilizarea metodei `getClass()`:

```
public class Object1 {
   public static void main(String args[]) {
      // crearea unui obiect de tip String
      String s = "Sir";
      // crearea unei referinte Object
      Object o = s;

      // afisarea numelui clasei din momentul executiei
      System.out.println(o.getClass().getName());
   }
}
```

Execuţia programului de mai sus va afişa la consolă:

```
java.lang.String
```

Chiar dacă apelul este realizat de o referinţă de tip `Object`, este afişat numele clasei efective a obiectului.

Fiecărei instanţe a unei clase i se asociază un cod unic pentru o execuţie a programului. Codul asociat se obţine cu metoda `hashCode()`. Metoda `equals()` testează dacă obiectul curent şi cel specificat au acelaşi cod. Dacă dorim ca pentru obiectele unei anumite clase să testeze altceva decât codul obiectului, atunci vom redefini metoda.

Exemplul 3.2.8. Testarea egalităţii codurilor aferente obiectelor:

```
public class Object2 {
  public static void main(String[] args) {
    // crearea a doua obiecte
    String s = "sir 1";
    Object o = new String("sir 2");

    // cele doua sunt diferite
    if (o.equals(s))
      System.out.println("Egalitate");
    else
      System.out.println("Diferenta");

    // acum devin identice
    o = s;
    // ambele sunt referinta la acelasi obiect
    if (o.equals(s))
      System.out.println("Egalitate");
    else
      System.out.println("Diferenta");
  }
}
```

Programul de mai sus va afişa:

```
Diferenta
Egalitate
```

Metoda `clone()` poate arunca excepţia `CloneNotSupportedException` în cazul în care clasa nu implementează interfaţa `Cloneable`. Aceasta nu conţine nici o metodă. De regulă, metoda `clone()` creează un nou obiect cu aceleaşi valori ale datelor membre, iar modificarea originalului sau a copiei nu va afecta şi celălalt obiect. Deoarece metoda `clone()` este protejată, aceasta nu poate fi apelată din exteriorul clasei. De obicei este apelată din interiorul claselor derivate.

Exemplul 3.2.9. Clonarea claselor se poate realiza în maniera următoare:

```
// definirea unei clase simple care permite clonarea
class ObiectSimplu implements Cloneable {
  // declararea unei date membre
  private String content = "";

  public void setContent(String s) {
    content=s;
  }
  public String getContent() {
    return content;
  }

  // supradefinirea metodei clone()
  public Object clone() {
    try {
      return super.clone();
    } catch (CloneNotSupportedException e) {
      System.err.println("Eroare la clonare!");
      return this;
    }
  }
}

public class Object3 {
  public static void main(String[] args) {
    // crearea unui obiect
    ObiectSimplu ob1 = new ObiectSimplu();
    // clonarea unui obiect
    ObiectSimplu ob2 = (ObiectSimplu) ob1.clone();

    // verficarea celor doua obiecte
    System.out.println(ob1.equals(ob2));
    System.out.println(ob1.getContent()==ob2.getContent());
    // modificarea originalului
    ob1.setContent("Test");
    // verficarea continutului celor doua obiecte
    System.out.println(ob1.getContent()==ob2.getContent());
  }
}
```

Programul anterior creează o copie a unui obiect, testează dacă într-adevăr s-a realizat o copie, după care se verifică dacă cele două obiecte sunt independente. Rezultatul va fi:

```
false
true
false
```

Prima linie denotă faptul că avem două obiecte, iar a doua că ambele au acelaşi conţinut. Ultima linie afişată arată că modificarea unui obiect nu are efect asupra celuilalt.

Metoda `toString()` este apelată automat ori de câte ori se cere o conversie la tipul `String`. Se utilizează frecvent în operaţia de concatenare a şirurilor de caractere.

Exemplul 3.2.10. Adăugăm la clasa `ObiectSimplu` metoda `toString()`:

```
  public String toString() {
    return "ObiectSimplu: " + content;
  }
```

Următorul program:

```
public class Object4 {
  public static void main(String[] args) {
    // crearea unui obiect
    ObiectSimplu ob = new ObiectSimplu();
    // setarea continutului
    ob.setContent("Test");
    // afisarea mesajului
    // metoda toString() va fi apelata implicit
    System.out.println(ob + "\t\t[ Ok ]");
  }
}
```

va afişa la execuţie:

```
ObiectSimplu: Test          [ Ok ]
```

3.2.5. Implementarea interfeţelor

O *interfaţă* este un tip special de dată care conţine un grup de date membre şi de metode abstracte. Fiecare clasă care implementează o interfaţă va trebui să redefinească respectivele metode.

Clauza opţională `implements` din declaraţia unei clase (vezi sintaxa generală în secţiunea 3.2) specifică lista interfeţelor imediat următoare clasei curente (este vorba de `<ListaInterfete>`, care se mai numesc *superinterfeţe*). Mai precis, în declaraţia `C1 implements I1, I2, ..., IN`, spunem că `I1`, `I2`, ..., `IN` sunt *superinterfeţe directe*. Acestea trebuie să fie accesibile, altfel se obţine eroare la compilare. Numele interfeţelor `I1`, `I2`, ..., `IN` trebuie să fie distincte, altfel se obţine eroare la compilare.

Dacă o clasă implementează o interfaţă, atunci respectiva clasă trebuie să redefinească toate metodele interfeţei (în caz contrar se obţine eroare la compilare). Există unele clase speciale numite *adaptori* care implementează o interfaţă definind toate metodele acesteia. De cele mai multe ori, acestea nu au nici un efect. Dacă doreşte să realizeze o clasă care să implementeze o interfaţă ce conţine mai multe metode, programatorul va extinde adaptorul corespunzător interfeţei şi va defini doar metodele care prezintă interes.

Ca şi în cazul claselor, relaţia de *derivare a interfeţelor* (indirectă) este închiderea tranzitivă a relaţiei de derivare directă a interfeţelor (vom reveni asupra interfeţelor în secţiunea 3.3).

Spunem că I_1 este *derivată* din I_N dacă există interfeţele I_2, I_3, ..., I_{N-1}, astfel încât I_1 este derivată direct din I_2, I_2 este derivată direct din I_3, ..., I_{N-1} este derivată direct din I_N. Interfeţele I_1, I_2, ..., I_N sunt distincte, în caz contrar, se obţine eroare la compilare.

Folosind relaţia de derivare a interfeţelor, spunem că un tip interfaţă `I` este superinterfaţă a tipului clasă `C` dacă `I` este interfaţă directă a lui `C` sau există `I'` superinterfaţă directă a lui `C`, astfel încât `I'` este derivată din `I`.

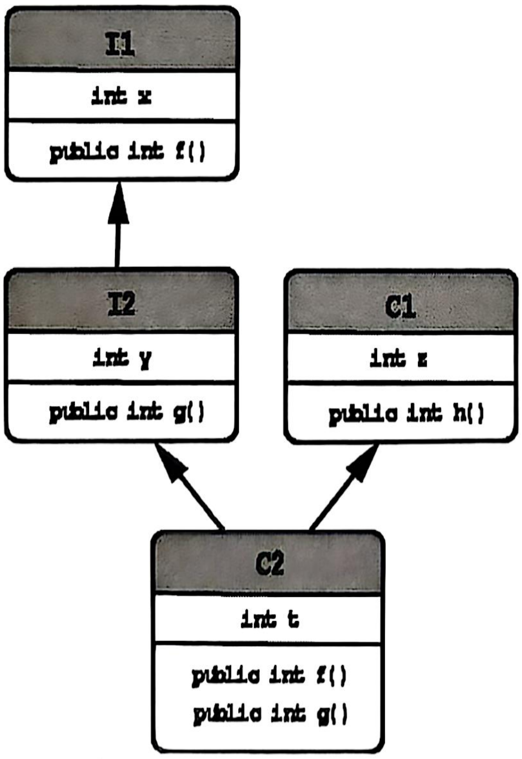
Exemplul 3.2.11. În programul Java de mai jos:

```
// declararea interfetei I1
interface I1 {
  // declararea unei date membre
  int x = 0;
  // declararea unei metode (care este abstracta)
  public int f();
}

// declararea interfetei I2
// aceasta va mosteni datele si metodele membre din I1
interface I2 extends I1 {
  int y = 1;
  public int g();
}

// declararea clasei C1
class C1 {
  int z = 2;
  public int h() { return z; };
}

/* declararea clasei C2 care extinde clasa C1
   si implementeaza interfata I2 */
class C2 extends C1 implements I2 {
  int t = 3;
  // redefinirea metodelor din interfata I2
  public int f() { return x; }
  public int g() { return y; }
}

public class SuperInterfete {
  public static void main(String args[]) {
    // instantierea unui obiect din clasa C2
    C2 obiect = new C2();
    System.out.println("x = " + obiect.f() +
      ", y = " + obiect.g() + ", z = " + obiect.h()
      + ", t = " + obiect.t);
  }
}
```

avem următoarele relaţii de derivare între clase şi interfeţe:

a) interfaţa `I2` este derivată din `I1`, adică `I1` este superinterfaţă pentru `I2`;
b) clasa `C2` este derivată din clasa `C1` şi din interfaţa `I2`, adică `C1` este superclasă a lui `C2` şi `I2` este superinterfaţă a lui `C2`;
c) conform cu a) şi b), rezultă că `C2` este derivată (indirect) din `I1`, adică `I1` este superinterfaţă (indirectă) pentru `C2`.

Această ierarhie de clase şi interfeţe este ilustrată şi în figura de mai jos:



Ierarhia de clase din exemplul 3.2.11

Este obligatoriu pentru `C2` (care nu este abstractă) să implementeze metodele `f()` şi `g()`; în caz contrar, se obţine eroare la compilare. Programul de mai sus va afişa la execuţie:

```
x = 0, y = 1, z = 2, t = 3
```

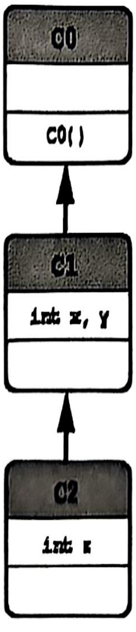
3.2.6. Declaraţiile membrilor unei clase

Corpul unei clase poate conţine declaraţii ale membrilor clasei, adică *atribute* şi *metode*. Corpul clasei poate conţine şi alte declaraţii, de nemembri, cum sunt iniţializatorii statici şi constructorii. Domeniul numelui unui membru declarat într-o clasă sau moştenit de o clasă este corpul declarat în acea clasă. Membrii moşteniţi ai unui tip clasă sunt membri moşteniţi din superclasa directă (cu excepţia lui `Object`, care nu are superclasă) şi membri moşteniţi din toate superinterfeţele directe. Membrii `private` ai unei clase nu sunt moşteniţi de alte subclase ale acelei clase. Doar membrii `protected` şi `public` sunt moşteniţi de subclase declarate în alt pachet. Deoarece iniţializatorii statici şi constructorii nu sunt membri, aceştia nu se moştenesc.

Dacă o clasă nu are nici un constructor, atunci se apelează implicit un constructor fără argumente pentru clasa curentă, care conţine un apel al constructorului implicit din superclasa acestuia. Acest lucru se realizează explicit prin intermediul cuvântului rezervat `super`, care este valid doar în metode, instanţe, constructori sau iniţializatori ai unei variabile, instanţă a unei clase. Doar în clasa `Object` nu se poate utiliza `super`, deoarece aceasta nu are superclase. Astfel, dacă avem `class C2 extends C1` şi clasa `C2` nu are constructori, atunci se creează un constructor implicit echivalent cu `C2(){super();}`. Dacă apelul lui `super()` într-un constructor este explicit, atunci acesta trebuie să fie prima instrucţiune în constructor. În caz contrar, se obţine eroare la compilare.

Exemplul 3.2.12. Următorul program Java pune în evidenţă accesul clasei derivate la clasa de bază folosind apelul implicit al lui `super()`. Dacă superclasa directă nu are nici ea constructor explicit, atunci se va apela constructorul implicit al acesteia ş.a.m.d., până la găsirea unei superclase care are constructor implicit sau fără argumente. În caz contrar, se obţine eroare la compilare.

Ierarhia de clase din acest exemplu este:



Ierarhia de clase din exemplul 3.2.12

```
class C0 {
  C0() {
    System.out.println("Am apelat constructorul C0()");
  }
}

class C1 extends C0 {
  int x = 1, y;
}

class C2 extends C1 {
  int z = 3;
}

public class TestDoi {
  public static void main(String[] args) {
    C2 obiect = new C2();
    System.out.println(obiect.x + obiect.y + obiect.z);
  }
}
```

La execuţie se va afişa:

```
Am apelat constructorul C0()
4
```

Clasa `C2` nu are constructor, deci se va crea constructorul implicit cu codul `C2() {super();}`, apelându-se constructorul lui `C1`. Deoarece nici această clasă nu are constructor, se va crea constructorul implicit cu codul `C1() { super();}`, apelându-se constructorul lui `C0`. Acum se va afişa mesajul `Am apelat constructorul C0()`. Valorilor atributelor `obiect.x`, `obiect.y` şi `obiect.z` sunt `1`, `0`, `3` (atributele neiniţializate explicit de programator se iniţializează cu 0). Astfel, se va afişa şi suma acestora.

Exemplul 3.2.13. Următorul program Java pune în evidenţă accesul clasei derivate la clasa de bază prin apelul explicit al lui `super()`.

```
// definirea clasei C1
class C1 {
  int x = 1;
  // definirea constructorilor
  C1() {
    f();
  }
  C1(int x) {
    this.x = x;
  }
```

```
    // definirea unei metodei
    void f() {
      System.out.print("x = " + x);
    }
  }

  // definirea clasei C2 care extinde C1
  class C2 extends C1 {
    int y;
    // definirea constructorului
    C2(int x, int y) {
      // apelarea explicita a constructorului superclasei
      super(x);
      this.y = y;
    }
    // definirea unei metode membre
    void g() {
      System.out.println("y = " + y);
      f();
    }
  }

  public class Test {
    public static void main(String[] args) {
      // crearea unui obiect de tip C2
      C2 obiect = new C2(2, 3);
      // invocarea metodei membre g()
      obiect.g();
    }
  }
```

La execuţie se va afişa:

```
y = 3
x = 2
```

La crearea unui obiect din clasa `C2` se va apela constructorul explicit cu două argumente, care, la rândul lui, prin intermediul lui `super()`, apelează constructorul clasei `C1` cu un singur parametru. Urmează afişarea celor două valori atribut.

3.2.6.1. *Niveluri de acces*

Există patru niveluri de acces: `public`, `protected`, implicit şi `private`.

Dacă tipul clasă sau interfaţă este declarat `public`, atunci acesta poate fi accesat de orice cod Java ce accesează pachetul în care este declarat. În caz contrar, acesta poate fi accesat doar din pachetul în care este declarat.

O dată sau o metodă membră a unui tip referinţă sau un constructor al unui tip clasă este accesibil doar dacă tipul este accesibil şi membrul sau constructorul permite accesul astfel:

a) Dacă membrul sau constructorul este declarat `public`, atunci se permite accesul. Toţi membrii unei interfeţe sunt implicit `public`;
b) Altfel, dacă membrul sau constructorul este declarat `protected`, atunci accesul este permis doar dacă:
   i. Accesul la membru sau constructor apare din pachetul clasei în care este declarat membrul `protected`;
   ii. Accesul apare dintr-o subclasă a clasei în care este declarat membrul `protected` (detalii mai jos);
c) Altfel, dacă membrul sau constructorul este declarat `private`, atunci accesul este permis doar dacă acesta apare din clasa în care este declarat;
d) Altfel, este vorba de *acces implicit* şi este permis doar dacă accesul apare din pachetul în care este declarat tipul.

Continuăm cu detalii despre modificatorul de acces `protected`. Un membru sau constructor `protected` al unui obiect poate fi accesat din afara pachetului în care este declarat doar de codul responsabil de implementarea acelui obiect. Astfel, fie `C1` clasa în care este declarat membrul sau constructorul `protected` şi fie `C2` subclasa în care apelăm/accesăm acest membru sau constructor `protected`. Atunci:

1. Dacă este vorba de un membru `protected`, pe care îl notăm cu `M`, avem:
   a) Dacă accesul se face printr-o expresie de forma `super.M`, atunci acesta este permis;
   b) Dacă accesul se face printr-un nume calificat `C.M`, unde `C` este o clasă, atunci accesul este permis doar dacă `C` este `C2` sau o subclasă a lui `C2`;
   c) Dacă accesul se face printr-un nume calificat `E.M`, unde `E` este o expresie, atunci accesul este permis doar dacă tipul expresiei `E` este `C2` sau o subclasă a lui `C2`;
   d) Dacă accesul se face printr-o expresie de acces de câmp `E.M`, unde `E` este o expresie primară sau un apel de metodă `E.M()`, atunci accesul este permis doar dacă `E` este `C2` sau o subclasă a lui `C2`.
2. Dacă este vorba de un constructor `protected`, avem:
   a) Dacă accesul se face printr-un apel de constructor al unei superclase (folosind `super()`), atunci accesul este permis;
   b) Dacă accesul se face printr-o expresie de creare a unei instanţe (folosind `new C()`), atunci accesul nu este permis;
   c) Dacă accesul se face printr-un apel al metodei `newInstance()` a clasei `Class`, atunci accesul nu este permis.

Exemplul 3.2.14. Exemplul de mai jos pune în evidenţă faptul că în exteriorul pachetului `P1`, clasa `C2` nu este accesibilă, nefiind declarată `public`. În schimb, atributul `x` şi metoda `f()` din clasa `C1` vor putea fi accesate din orice pachet, deoarece sunt declarate `public`.

```
package P1;

public class C1 {
  public int x = 1;
  public void f(int x) {
    this.x += x;
  }
}

class C2 {
  C1 obiectC1;
}
```

Astfel, următorul program va accesa, din afara pachetului `P1`, atributul şi metoda clasei `C1`. Dacă am fi încercat crearea unui obiect din clasa `C2`, am fi obţinut eroare la compilare, deoarece clasa `C2` nu este declarată public, deci nu există acces în afara pachetului `P1`.

```
import P1.*;

public class Test {
  public static void main(String[] args) {
    C1 obiect = new C1();
    obiect.f(2);
    System.out.println(obiect.x);
  }
}
```

Programul de test este corect şi va afişa la execuţie `3`.

Continuăm cu un exemplu în care evidenţiem accesul la atribute `protected`.

Exemplul 3.2.15. Fie următorul program care face parte din pachetul `p1`:

```
package p1;

public class C1 {
  protected int x = 2;
  public void f(int x) {
    this.x += x;
  }
  public int getX() {
    return x;
  }
}
```

Clasa `C1` este declarată `public`, deci este accesibilă şi din afara pachetului `p1`. Metodele `f()` şi `getX()` sunt declarate `public`, deci sunt accesibile tuturor obiectelor de tip `C1`. Atributul `x` este declarat `protected`, deci este accesibil doar subclaselor clasei `C1`, acesta putând fi accesat (citit/modificat) doar prin intermediul funcţiilor de acces (`f()` şi `getX()`). Astfel, programul:

```
import p1.*;
public class Test {
  public static void main(String[] args) {
    C1 obiect = new C1();
    obiect.f(2);
    System.out.println(obiect.getX());
  }
}
```

va afişa la execuţie `4`.

Un exemplu în care punem în evidenţă accesul implicit la atribute este următorul:

Exemplul 3.2.16. Fie următoarele programe care fac parte din pachetul `p1`:

```
package p1;

public class C1 {
  protected int x = 2;
  public void f(int x) {
    this.x += x;
  }
  public void g() {
    System.out.println("x = " + x);
  }
}
```

şi, respectiv clasa `C2`, derivată a lui `C1`, din pachetul `p1`:

```
package p1;

public class C2 extends C1 {
  int y = 2;
  public void f(int x, int y) {
    this.x += x;
    this.y += y;
  }
  public void g() {
    super.g();
    System.out.println("y = " + y);
  }
}
```

Considerăm clasa `C3` derivată a lui `C2`. Clasa `C3` nu face parte din pachetul `p1`. Accesul la metoda `f()` din clasa `C2` se face folosind cuvântul rezervat `super`. Dacă se încearcă accesarea directă (cum am accesat atributul `x` în clasa `C2`) a metodei `f()`, atunci s-ar obţine eroare la compilare.

```
import p1.C2;

class C3 extends C2 {
  int z = 3;
  public void f(int x, int y, int z) {
    super.f(x, y);
    this.z += z;
  }
  public void g()
    super.g();
    System.out.println("z = " + z);
  }
}
```

Considerăm acum şi clasa principală a aplicaţiei, care face parte din acelaşi pachet cu clasa `C3`:

```
import p1.*;

public class Test {
  public static void main(String[] args) {
    C3 obiect = new C3();
    obiect.f(2, 3, 4);
    obiect.g();
  }
}
```

care va afişa la execuţie:

```
x = 3
y = 5
z = 7
```

Exemplul 3.2.17. Evidenţierea accesului la atribute `private`:

```
class C1 {
  private int x = 1;
  void f(int x) {
    this.x += x; n++;
  }
  private static int n;
  void afisareN() {
    System.out.println(n);
  }
}

class C2 extends C1 {
  private int y = 2;
  void f(int x, int y) {
    super.f(x);
    this.y += y;
//    n++;
  }
}

public class Test {
  public static void main(String[] args) {
    C2 obiect = new C2();
    obiect.f(2, 3);
    obiect.afisareN();
  }
}
```

Accesarea lui `n` în clasa `C2` conduce la eroare la compilare, deoarece atributul `n` al clasei `C1` este `private`, deci `n` este accesibil doar în clasa `C1`. Execuţia programului de mai sus va afişa `1`.

3.2.6.2. *Declaraţiile atributelor unei clase*

`<DeclaratieAtribut>` (din definiţia declarării unei clase, secţiunea 3.2) are sintaxa generală:

```
[<ModificatorAtribut>] <Tip> <DeclaratieVariabila1> [, . . . ,
   <DeclaratieVariabilaN>];
```

unde:

1. `<ModificatorAtribut>` poate fi: `public`, `protected`, `private`, `final`, `static`, `transient`, `volatile`;
2. `<DeclaratieVariabilaI>`, cu `I` = 1, ..., `N`, poate fi:
   a) `<Identificator>` [= `<InitializatorVariabila>`] sau
   b) `<Identificator>` `[]...[]` [= `<InitializatorTablou>`]

În cazul declarării unui atribut de tip tablou, nu trebuie confundate parantezele pătrate (`[]`) cu convenţia de notaţie a unei părţi opţionale. `<Identificator>` va fi numele atributului declarat. Numele unui atribut are domeniul în întreg corpul clasei în care este declarat. Într-o declaraţie a unei clase, atributele trebuie să aibă nume distincte, în caz contrar se obţine eroare la compilare. Atributele şi metodele pot avea acelaşi nume, din moment ce ele se utilizează în contexte diferite. Dacă o clasă declară un atribut cu un anumit nume, se spune că acel atribut *ascunde* toate declaraţiile atributului cu acelaşi nume din superclasele şi superinterfeţele acelei clase. O clasă moşteneşte din superclasa directă şi din superinterfeţele directe toate atributele superclasei şi superinterfeţelor care sunt accesibile codului clasei şi nu sunt ascunse în declaraţia clasei. Este posibil ca o clasă să moştenească mai multe atribute cu acelaşi nume. Un atribut ascuns poate fi accesat prin nume calificat (dacă acesta este `static`), printr-o expresie de acces a atributului folosind cuvântul rezervat `super` sau prin conversie explicită la tipul superclasei.

Exemplul 3.2.18. Programul Java de mai jos pune în evidenţă câteva dintre modalităţile de acces la atributele ascunse ale unei clase. Se observă că atributul `x` şi metoda `x()` sunt declarate atât în clasa `C1`, cât şi în clasa `C2`. Acestea sunt accesate în clasa `TestAtribute` fără a se obţine eroare la compilare. Atributul static `y` poate fi accesat prin adăugarea numelui clasei la care ne referim.

```
class C1 {
  private int x = 1;
  public static int y = 3;
  void x() {
    System.out.println(x);
  }
}

class C2 extends C1 {
  private int x = 2;
  public static int y = 4;
  void x() {
    super.x();
    System.out.println(x);
  }
}

public class TestAtribute {
  public static void main(String[] args) {
    C2 obiect = new C2();
    obiect.x();
    System.out.println("C1.y = " + C1.y);
    System.out.println("C2.y = " + C2.y);
  }
}
```

Execuţia acestui program va afişa:

```
1
2
C1.y = 3
C2.y = 4
```

Continuăm cu prezentarea modificatorilor atributelor. În secţiunea 3.2.5.1, au fost prezentaţi modificatorii `public`, `private`, `protected`, precum şi cel implicit.

Dacă un atribut este declarat `static`, atunci se alocă o singură dată memorie pentru acesta, indiferent de numărul de instanţe ale clasei respective. Deoarece toate instanţele unei clase au acelaşi atribut (cu aceeaşi valoare în orice moment al execuţiei programului), atributele statice se mai numesc *variabile clasă*. În contrast, un atribut nestatic se numeşte ***variabilă instanţă***. Când se creează o nouă instanţă a clasei, se creează şi o nouă variabilă asociată atributului nestatic. Atributele statice sunt iniţializate în momentul încărcării clasei, deci înaintea creării instanţelor acelei clase.

Exemplul 3.2.19. Fie declarația clasei `Increment`:

```
class Increment
{
   static int x = 0;
   Increment() { x ++; } // constructorul
}
```

Variabila `x` este statică. Asta înseamnă că oricâte instanţe ale clasei `Increment` sunt create, există doar un `x` (pentru care s-a alocat memorie doar o singură dată). De altfel, şi iniţializarea cu `0` are loc în momentul încărcării clasei. Variabila statică este incrementată de fiecare dată când se apelează constructorul clasei, astfel fiind posibil să ştim câte obiecte ale clasei `Increment` s-au creat.

Există două moduri de referire la o variabilă statică:

1. printr-o referinţă la orice obiect al clasei;
2. prin numele clasei (aceasta este varianta recomandată datorită clarităţii codului).

Exemplul 3.2.20. Continuăm exemplul 3.2.19 punând în evidenţă eventualele confuzii care pot apărea dacă utilizăm prima formă:

```
Increment e1 = new Increment();
Increment e2 = new Increment();
System.out.println("e1.x = " + e1.x);
e1.x = 100;
e2.x = 200;
System.out.println("e1.x = " + e1.x);
```

Prima dată se afişează `e1.x = 2`, iar a doua oară `e1.x = 200`. Justificarea constă în faptul că obiectele `e1` şi `e2` (instanţe ale clasei `Increment`) au aceeaşi zonă de memorie pentru atributul static `x`. Cu alte cuvinte, `e1.x` şi `e2.x` se referă la aceeaşi variabilă (statică).

Atributele statice au multe aplicaţii, cum ar fi numărarea instanţelor unei clase.

Exemplul 3.2.21. Reluăm exemplul 3.2.19 pentru a crea o modalitate de numărare a instanţelor unei clase. Astfel, declarăm un atribut static `numarInstante`, care va fi incrementat în constructor. La fiecare instanţiere a clasei `Increment`, se va apela constructorul `Increment()` şi deci se va incrementa variabila clasă `numarInstante`.

```
public class NumarareInstante {
  public static void main(String args[]) {
   System.out.println("S-au creat " + Increment.numarInstante +
      " instante ale clasei Increment");
   Increment a = new Increment();
   System.out.println("S-au creat " + Increment.numarInstante +
      " instante ale clasei Increment");
   Increment b = new Increment();
```

```
      System.out.println("S-au creat " + Increment.numarInstante +
        " instante ale clasei Increment");
    }
}

class Increment {
  static int numarInstante = 0;
  Increment() {
    numarInstante++;
  };
}
```

La execuţie, acest program va afişa:

```
S-au creat 0 instante ale clasei Increment
S-au creat 1 instante ale clasei Increment
S-au creat 2 instante ale clasei Increment
```

Prezentăm în continuare modificatorul de atribut `final`. Declaraţiile atributelor `final` (atât ale variabilelor clasă, cât şi ale celor instanţă) trebuie să conţină o expresie de iniţializare, în caz contrar, apare o eroare de compilare. Valoarea unui atribut `final` este aceeaşi pe toată execuţia programului. Orice asignare ulterioară a unui atribut `final` conduce la o eroare de compilare. Dacă un atribut `final` va fi egal cu o referinţă la un obiect, atunci se poate schimba starea obiectului, dar atributul se va referi mereu la acelaşi obiect. Declararea unui atribut `final` poate servi la documentarea programului, la evitarea erorilor de programare şi poate face mai uşoară pentru compilator generarea codului.

Exemplul 3.2.22. Următorul program pune în evidenţă utilizarea unui atribut `final`.

```
class C1 {
  int x;
  C1(int x) {
    this.x = x;
  }
  void setX(int x) {
    this.x = x;
  }
  final static C1 y = new C1(1);
}

public class AtributFinal {
  public static void main(String args[]) {
    System.out.println(C1.y);
    C1 obiect = new C1(4);
    System.out.println(obiect.y);
    obiect.setX(6);
```

```
        System.out.println(obiect.y);
    }
}
```

Atributul y trebuie declarat şi `static` pentru că altfel, la crearea primului obiect din clasa `C1`, se va crea o instanţă a clasei şi pentru atributul y al clasei `C1`, care, la rândul ei, va genera iar o instanţă asociată atributului y ş.a.m.d. Astfel, execuţia programului va intra într-un fel de „recursie" şi va umple memoria de instanţe ale clasei `C1`. În schimb, declarând atributul y ca fiind `static`, vom avea o singură valoare pentru variabila clasă y pentru toate instanţele clasei `C1`. Programul va afişa la execuţie un rezultat de genul (la execuţii în momente de timp diferite se pot obţine alte adrese de memorie):

```
C1@3179c3
C1@3179c3
C1@3179c3
```

Astfel, chiar dacă se schimbă starea obiectului (modificând valoarea atributului x), valoarea atributului y va fi mereu aceeaşi.

Variabilele descrise `final` sunt, de fapt, constante.

Exemplul 3.2.23. Un exemplu de definire a unui atribut constant este cel utilizat pentru transformarea incilor în centimetri.

```
final float CmInch = 2.54f;
```

Reamintim faptul că litera `f` de la sfârşitul numărului indică faptul că acesta este considerat de tip `float`.

Variabilele `final  static` sunt folosite pentru declararea constantelor care sunt utilizate într-un număr de clase diferite într-un program.

Exemplul 3.2.24. Putem considera o clasă care conţine doar constante, astfel:

```
public class Constante {
  public final static float C=3.0E+8f;
  public final static int LATIME=640;
     ...
}
```

Prin convenţie, numele constantelor se scriu cu majuscule.

Exemplul 3.2.25. Constanta `PI` este declarată în clasa `Math` astfel:

```
public final static double PI;
```

Continuăm cu prezentarea variabilelor `transient`. Variabilele marcate `transient` nu fac parte din starea persistentă a unui obiect. Acestea se utilizează la serializarea obiectelor. *Serializarea* este procesul de împărţire a datelor în octeţi, astfel încât să poată fi trimişi printr-un flux de ieşire. La implementarea unei clase trebuie ţinut cont care parte a stării clasei trebuie serializată. Implicit, toate datele nestatice şi neocazionale (non-transient) vor fi serializate. Aceasta înseamnă că, dacă nu se doreşte serializarea unor anumite variabile, atunci acestea trebuie declarate `transient`. Vom reveni asupra modificatorului `transient` în capitolul 5.

Exemplul 3.2.26. În cadrul clasei `C1` de mai jos:

```
class C1 {
  int x;
  transient int y;
}
```

doar atributul `x` va putea fi salvat şi serializat, deoarece nu este declarat `transient`.

Prezentăm mai jos variabilele `volatile`. Limbajul Java permite firelor de execuţie care accesează variabile să păstreze copii de lucru ale acestora; aceasta permite o implementare mai eficientă a firelor de execuţie multiple. Copiile de lucru trebuie să fie corelate cu originalele din memoria principală numai în punctele de sincronizare prescrise, adică la blocarea (engl. *lock*) şi deblocarea obiectelor (engl. *unlock*). Ca o regulă, pentru a fi siguri că aceste variabile partajate sunt consistente şi actualizate, un fir de execuţie trebuie să se asigure că are drept de folosire exclusivă a acestor variabile prin blocarea lor.

Modificatorul `volatile` specifică faptul că o variabilă poate să fie modificată în orice moment de către firele de execuţie. Scopul modificatorului `volatile` este protecţia împotriva coruperii memoriei unei variabile. Într-un mediu asincron, coruperea datelor apare când variabila este memorată în regiştrii procesorului. Modificatorul `volatile` determină sistemul de execuţie Java să refere o variabilă direct din memorie, în loc de a utiliza regiştri. În plus, variabila este citită din memorie şi scrisă înapoi în memorie după fiecare acces la memorie. Modificatorul `volatile` se utilizează foarte rar, preferându-se în schimb utilizarea lui `synchronized`.

Java pune la dispoziţie posibilitatea declarării atributelor `volatile`, caz în care copiile de lucru trebuie să fie corelate cu originalele din memoria principală de fiecare dată când se accesează variabila. Mai mult, operaţiile asupra originalelor pentru variabilele `volatile` sunt procesate de memoria principală în exact aceeaşi ordine ca şi cea cerută de firul de execuţie. Vom reveni cu detalii în capitolul dedicat firelor de execuţie.

Exemplul 3.2.27. Considerăm clasa `C1`, care conţine variabilele `i`, `j` ce sunt incrementate în metoda `f()`. Metoda `afisare()` va tipări valoarea acestora. Clasa `C2` extinde clasa `Thread` (referitoare la fire de execuţie; vom reveni în capitolul 5) şi conţine în metoda `run()` o structură de tip `for` care va realiza 100 de iteraţii. Clasa `C3` creează în metoda `main()` două fire de execuţie care vor apela metoda `f()`, respectiv `afisare()`.

```
// declararea clasei C1
// aceasta contine doar membri statici
class C1 {
  static int i = 0, j = 0;
  static void f() {
    i++;
    j++;
  }
  static void afisare() {
    System.out.print("i = " + i);
    System.out.println(" j =" + j);
  }
}

// se declara clasa corespunzatoare unui fir de executie
class C2 extends Thread {
  private C1 c1;
  private String numeFir;

  public C2(C1 c1, String numeFir) {
    this.c1 = c1;
    this.numeFir = numeFir;
  }

  // metoda run() se executa la pornirea firului de executie
  public void run() {
    for (int i = 1; i <= 100; i++) {
      System.out.println(numeFir + " este la pasul " + i);
      if (numeFir.equals("FirUnu")) c1.f();
      else c1.afisare();
    }
  }
}
public class C3 {
  public static void main(String args[]) {
    C1 obiect = new C1();
    // se creeaza doua fire de executie
    C2 fir1 = new C2(obiect, "FirUnu");
    C2 fir2 = new C2(obiect, "FirDoi");
    // se pornesc firele de executie
    // metoda start() se termina imediat
    fir1.start();
    fir2.start();
  }
}
```

La execuţia acestui program, pot interveni situaţii când metoda `afiseaza()` va tipări o valoare mai mare pentru `j` decât valoarea lui `i`, deoarece exemplul nostru nu conţine mecanisme de sincronizare. Astfel, se poate afişa ceva de tipul:

```
FirDoi este la pasul 8
i=37 j=37
FirDoi este la pasul 9
i=37FirUnu este la pasul 39
 j=38
FirDoi este la pasul 10
i=38 j=38
```

O primă modalitate de prevenire a acestui lucru este adăugarea în lista modificatorilor metodelor `f()` şi `afiseaza()` a cuvântului rezervat `synchronized` (vom reveni asupra acestuia în capitolul 6). Aceasta va preveni ca metodele `f()` şi `afiseaza()` să poată fi apelate de cel mult un fir de execuţie (neconcurent) şi este clar că valorile divizate ale lui `i` şi `j` vor fi actualizate înaintea returnării din metoda `f()`. Aşadar, valoarea lui `j` nu va fi niciodată mai mare decât cea a lui `i` în cadrul metodei `afiseaza()`.

A doua modalitate este să declarăm atributele `x` şi `y` din clasa `C1` ca fiind `volatile`. Aceasta va permite ca metodele `f()` şi `afiseaza()` să fie executate concurent, dar garantează accesarea valorilor divizate pentru `i` şi `j` exact în acelaşi timp şi în aceeaşi ordine cum apar în program. Astfel, metoda `afiseaza()` nu va afişa valoarea lui `j` mai mare decât a lui `i`.

Continuăm acum cu iniţializarea atributelor, care, în principiu, au semantica unei instrucţiuni de asignare ataşate unei variabile declarate, astfel:

1. Dacă declaraţia se referă la o variabilă clasă (adică atribut static), atunci iniţializatorul variabilei este evaluat şi se face o singură dată asignarea atunci când este iniţializată clasa;
2. Dacă declaraţia se referă la o variabilă instanţă (adică atribut nestatic), atunci iniţializatorul variabilei este evaluat şi se face asignarea de fiecare dată când se creează o instanţă a clasei.

În ceea ce priveşte o expresie de iniţializare a unei variabile clasă, aceasta nu trebuie să utilizeze variabile a căror valoare nu se cunoaşte sau chiar variabila însăşi.

Exemplul 3.2.28. Fie clasa de mai jos:

```
class EroareInitializareVariabileClasa {
   static int a = b; // initializarea 1
   static int b = 3; // initializarea 2
   static int c = c + 3; // initializarea 3
}
```

Iniţializarea 1 este greşită (şi va conduce la eroare la compilare), deoarece variabila `b` nu are valoarea cunoscută la momentul iniţializării lui `a`. Iniţializarea 2 este corectă, în schimb iniţializarea 3 este greşită (şi va conduce la eroare la compilare), deoarece variabila `c` nu are valoarea cunoscută la momentul iniţializării sale.

Dacă o referinţă a unui nume simplu al oricărei variabile instanţă sau cuvintele rezervate `this` şi `super` apar într-o expresie de iniţializare pentru o variabilă clasă, atunci apare o eroare la compilare. Explicaţia constă în faptul că atributele statice se iniţializează în momentul încărcării claselor, deci înaintea creării de instanţe pentru acestea.

Ca şi în cazul variabilelor clasă, în ceea ce priveşte o expresie de iniţializare a unei variabile instanţă, aceasta nu trebuie să folosească variabile a căror valoare nu se cunoaşte sau chiar variabila însăşi. În schimb, o expresie de iniţializare a unei variabile instanţă poate conţine un nume de variabilă statică, precum şi cuvintele rezervate `this` şi `super`.

Exemplul 3.2.29. Fie programul Java de mai jos:

```
class C1 {
  int a = b + 1;        // initializarea 1
  static int b = 3;     // initializarea 2
  int c = this.a + 5;   // initializarea 3
}

public class InitializareVariabileInstanta {
  public static void main(String args[]) {
    C1 obiect = new C1();
    System.out.println("a = " + obiect.a + ", b = " +
      obiect.b + ", c = " + obiect.c);
  }
}
```

Iniţializarea 1 este corectă deoarece variabila `b` are valoarea cunoscută la momentul iniţializării lui `a`. Evident, iniţializarea 2 este corectă. Iniţializarea 3 este corectă deoarece variabila `a` are valoarea cunoscută la momentul iniţializării atributului `c`. Execuţia programului Java de mai sus va afişa:

```
a = 4, b = 3, c = 9
```

Continuăm cu prezentarea unor exemple de declaraţii de atribute. Referirea unui atribut static dintr-o clasă de bază se poate realiza fie prin cuvântul rezervat `super`, fie prin numele clasei.

Exemplul 3.2.30. Fie programul Java de mai jos:

```
class C1 {
  static int x = 4;
}

class SupraScriereAtributeStatice extends C1 {
  static double x = 2.3;
  public static void main(String[] args) {
    new SupraScriereAtributeStatice().afisare();
  }
```

```
   void afisare() {
      System.out.println(x + " " + super.x);
      System.out.println(SupraScriereAtributeStatice.x + " " +
        C1.x);
   }
}
```

Atributul static `x` apare atât în clasa `SupraScriereAtributeStatice`, cât şi în clasa de bază `C1`. Astfel, atributul `x` din clasa `C1` este suprascris (sau ascuns) de către cel din clasa `SupraScriereAtributeStatice`. Programul va afişa la execuţie:

```
2.3 4
2.3 4
```

În ceea ce priveşte suprascrierea (ascunderea) atributelor instanţă, există (cel puţin) trei modalităţi de accesare a acestora:

1. utilizând cuvintele rezervate `this` (care se referă la obiectul curent) şi `super` (care se referă la atributele din clasa de bază a obiectului curent);
2. utilizând o conversie explicită la tipul de bază;
3. creând câte un obiect pentru fiecare clasă în parte.

Exemplul 3.2.31. Programul Java de mai jos pune în evidenţă toate cele trei situaţii discutate:

```
class C1 {
   int x = 4;
}

class SupraScriereAtributeInstanta extends C1 {
   double x = 2.3;
   public static void main(String[] args) {
      SupraScriereAtributeInstanta obiectUnu =
        new SupraScriereAtributeInstanta();
      obiectUnu.afisare();
      System.out.println(obiectUnu.x + " " + ((C1)obiectUnu).x);
      C1 obiectDoi = new C1();
      System.out.println(obiectUnu.x + " " + obiectDoi.x);
   }
   void afisare() {
      System.out.println(this.x + " " + super.x);
   }
}
```

Se va afişa la execuţie:

```
2.3 4
2.3 4
2.3 4
```

O clasă poate moşteni două sau mai multe atribute cu acelaşi nume de la două interfeţe sau de la o superclasă şi o interfaţă. Pentru a evita erorile de compilare care pot apărea din cauza ambiguităţii numelui atributului, se poate utiliza un nume calificat sau o expresie de acces care foloseşte cuvântul rezervat `super`.

Exemplul 3.2.32. Programul Java de mai jos pune în evidenţă rezolvarea ambiguităţii unui atribut cu acelaşi nume dintr-o clasă, respectiv o interfaţă:

```
interface I1 {
  float x = 2.3f;
}

class C1 {
  int x = 4;
}

public class SuprascriereMultipla extends C1 implements I1 {
  public static void main(String[] args) {
    new SuprascriereMultipla().afisare();
  }
  void afisare() {
    System.out.println(super.x + " " + I1.x);
  }
}
```

Expresia de acces `super.x` se va referi la atributul `x` din clasa `C1`, iar numele calificat `I1.x` se va referi la atributul `x` al interfeţei `I1`. La execuţie se va afişa:

```
4 2.3
```

Dacă un atribut este moştenit prin mai multe căi dintr-o interfaţă, acesta se consideră că este moştenit o singură dată şi se poate accesa prin numele său simplu fără ambiguitate.

Exemplul 3.2.33. Programul Java de mai jos pune în evidenţă rezolvarea ambiguităţii unui atribut cu acelaşi nume moştenit de două ori dintr-o interfaţă:

```
interface I1 { int x = 2; }
interface I2 extends I1 { }
class C1 { }
class C2 extends C1 implements I1 { }

public class C3 extends C2 implements I2
{
  public static void main(String args[]) {
    System.out.println("x = " + x);
  }
}
```

Atributul x este moştenit şi din clasa `C2` (prin implementarea interfeţei `I1`), cât şi din interfaţa `I2` (prin derivarea din `I1`). Programul va afişa la execuţie:

```
x = 2
```

3.2.6.3. *Declaraţiile metodelor unei clase*

O metodă declară cod executabil care poate fi executat trimiţând un număr fix sau variabil de valori ca argumente. `<DeclaratieMetoda>` (din definiţia declarării unei clase, secţiunea 3.2) are sintaxa generală:

```
[<ModificatorMetoda>] <TipRezultat> <Identificator>
([<ListaParametriFormali>])
throws <TipClasa1> [, . . ., <TipClasaN>] <CorpMetoda>
```

unde:

1. `<ModificatorMetoda>` poate fi: `public`, `protected`, `private`, `abstract`, `final`, `static`, `synchronized`, `native` (fiecare modificator poate apărea cel mult o dată, iar dintre primii trei modificatori poate apărea cel mult unul; în caz contrar, se obţine eroare la compilare);
2. `<TipRezultat>` poate fi `void` sau un tip Java (primitiv sau referinţă);
3. `<Identificator>` este numele metodei;
4. `<ListaParametriFormali>` poate lipsi, dar dacă este prezentă, atunci are forma:

   ```
   <Tip1> <Identificator1> []...[], ..., <TipN> <IdentificatorN>
   []...[] [, <TipA> ... <IdentificatorA>]
   ```

   unde identificatorii reprezintă parametri formali, iar dacă se referă la variabile de tip tablou, atunci vor conţine paranteze pătrate `[]`, altfel acestea vor lipsi;
5. `<CorpMetoda>` poate fi un bloc de metodă sau nimic.

Numele metodei (`<Identificator>`) poate fi folosit pentru referirea către acea metodă şi poate coincide cu numele clasei sau cu un atribut al clasei. Dacă există deja definită o metodă cu aceeaşi *signatură* (nume, număr de parametri şi tipurile parametrilor), atunci se obţine o eroare la compilare. Metodele şi atributele pot avea acelaşi nume, pentru că se utilizează în contexte diferite. Parametrul formal este cel care apare în definiţia (prototipul) unei metode. La apelul acesteia, parametrii sunt actuali, iar valorile acestora sunt transmise parametrilor formali.

Nu este posibil ca doi parametri formali să aibă acelaşi nume (se obţine eroare la compilare). Când se apelează o metodă, valorile expresiilor parametrilor actuali vor iniţializa variabilele parametru nou create. Domeniul de vizibilitate a numelor parametrilor formali este întreg corpul metodei. Aceste nume de parametri nu pot fi redeclarate ca variabile locale sau parametri excepţie în metodă (deci nu este permisă ascunderea numelor parametrilor).

Exemplul 3.2.34. Următoarea clasă va produce o eroare la compilare, deoarece metoda `f()` nu poate fi supraîncărcată (cele două definiţii ale lui `f()` au aceeaşi signatură, diferă doar tipul întors):

```
class C1  {
  int x;
  int f(int x) { return x; }
  void f(int x) { x = this.x; }
}
```

Continuăm cu prezentarea modificatorilor metodelor. Dacă există mai mulţi modificatori, atunci se preferă ordinea menţionată în definiţia de mai sus.

Începem prezentarea modificatorului `abstract`. O declaraţie a unei metode abstracte specifică faptul că acea metodă este membru al clasei, furnizându-i signatura, tipul returnat şi clauzele `throws`, dar fără a descrie o implementare a sa. Dacă o metodă este declarată `abstract`, atunci şi clasa din care face parte trebuie declarată `abstract`, altfel apare o eroare la compilare. Fiecare subclasă care nu este declarată `abstract` trebuie să furnizeze o implementare a metodei abstracte, altfel apare o eroare la compilare.

Dacă o metodă conţine modificatorul `abstract`, atunci nu mai poate conţine modificatorii: `private`, `static`, `final`, `native` sau `synchronized`. Este imposibil să se implementeze o metodă `private abstract`, deoarece metodele `private` nu sunt accesibile în subclase. De asemenea, este imposibil să se implementeze o metodă `abstract final`, deoarece metodele `final` nu mai pot fi suprascrise. Este imposibil să se implementeze o metodă `abstract static`, deoarece metodele statice trebuie să fie implementate la încărcarea claselor.

Exemplul 3.2.35. O metodă instanţă care nu este declarată `abstract` poate fi suprascrisă de o metodă declarată `abstract`.

```
class C1 {
  public String f() {
    // returneaza numele clasei
    return this.getClass().getName();
  }
}

// definirea unei clase abstracte
abstract class C2 extends C1 {
  int x = 2;
  // supradefinirea metodei f() mostenita din C1
  public abstract String f();
  protected String g() {
    // se apeleaza metoda f() din clasa C1
    return super.f() + ", x = " + x;
  }
}

// se declara clasa C3 care nu este abstracta
class C3 extends C2 {
    int y = 3;
    // se redefineste metoda f() mostenita din C2
    public String f() {
      return g() + ", y = " + y;
    }
}
public class TestAbstract {
    public static void main(String args[]) {
      C3 obiect = new C3();
      // se apeleaza metoda f() din clasa C3
      System.out.println(obiect.f());
    }
}
```

Metoda `f()` din clasa `C2` suprascrie o metodă deja implementată în clasa `C1`. Metoda `g()` din clasa `C2` apelează metoda `f()` din clasa `C1` al cărei cod există. În metoda `g()` este esenţială prezenţa cuvântului rezervat `super`, în caz contrar s-ar fi obţinut eroare la compilare, deoarece s-ar fi apelat metoda `f()` a clasei `C2`, al cărei cod lipsea. Clasa `C3` trebuie să conţină definiţia metodei `f()`, în caz contrar s-ar fi obţinut eroare la compilare, pentru că `C3` nu a fost declarată `abstract`. În clasa de aplicaţie `TestAbstract`, se apelează pentru `obiect` din clasa `C3` metoda `f()`, care la rândul ei apelează metoda `g()` din clasa `C2` şi care apelează în sfârşit metoda `f()` din clasa `C1`. Astfel, programul Java de mai sus va afişa:

```
C3, x = 2, y = 3
```

O metodă declarată `final` nu poate fi suprascrisă (adică redefinită într-o clasă derivată) sau ascunsă (declarată `private`), în caz contrar se obţine eroare la compilare. O metodă declarată `private` şi toate metodele declarate într-o clasă `final` sunt implicit finale, deoarece este imposibilă suprascrierea acestora. De asemenea, este o eroare la compilare ca o metodă să fie declarată concomitent `final` şi `abstract`.

O metodă declarată `static` se numeşte *metodă a clasei* şi poate fi apelată fără referinţă la un obiect particular. O încercare de referire la obiectul curent folosind `this` sau `super` în corpul metodei clasei conduce la o eroare de compilare. O metodă care nu este declarată `static` se numeşte *metodă instanţă* (sau metodă nestatică). O metodă instanţă se apelează în raport cu un obiect, numit *obiect curent*, care poate fi referit cu `this` sau `super`. O metodă statică nu poate fi declarată `abstract`, în caz contrar apare o eroare la compilare.

O metodă `synchronized` (sincronizată) trebuie să capete dreptul de exclusivitate înainte de execuţia acesteia, în sensul că nici un alt fir de execuţie nu va executa în acel moment respectiva metodă. Detalii asupra metodelor sincronizate vor fi descrise în capitolul 6.

Clasa `java.lang.Math` conţine metoda:

```
public static synchronized double random();
```

care returnează un număr aleator de tip `double` în „intervalul" [0.0, 1.0). Această metodă este declarată `synchronized` pentru a permite utilizarea sa corectă de mai multe fire de execuţie (astfel încât calculul următorului număr pseudoaleator să fie realizat corect).

Continuăm cu prezentarea metodelor declarate `native`. O metodă Java este ***nativă*** dacă este definită în alt limbaj de programare (de exemplu, C, C++) şi este doar declarată într-un program Java de unde este apelată. O metodă declarată `native` nu poate fi declarată şi `abstract`, deoarece codul acesteia se defineşte în biblioteca dinamică, şi nu într-o subclasă. Una dintre trăsăturile importante ale limbajului Java este independenţa de platformă, adică codul compilat pe o platformă poate să ruleze pe orice altă platformă care are un interpretor Java. Folosind metode native, acest avantaj se pierde, deoarece aplicaţiile vor rula doar pe platformele aceluiaşi sistem de operare şi folosind aceleaşi biblioteci dinamice capabile să memoreze metodele native ale aplicaţiei. Java posedă *interfaţa nativă Java* (engl. *Java Native Interface* – JNI) care permite apelul metodelor native dintr-o aplicaţie Java.

Exemplul 3.2.36. În cele ce urmează, vom prezenta o aplicaţie simplă Java ce apelează o metodă nativă scrisă în C, care la rândul ei apelează funcţia `MessageBox()` din Win32 API (engl. *Windows 32 Application Programming Interface*) pentru a afişa o fereastră de dialog conţinând un text. Sub platforma Windows, trebuie urmaţi cinci paşi:

Pasul 1. Scrierea programului Java (numit `AfisareMesajJNI.java`) care declară metoda nativă C:

```
public class AfisareMesajJNI {
  public static void main(String [] args) {
    AfisareMesajJNI obiect = new AfisareMesajJNI();
    obiect.afisareMesaj("Text afisat folosind JNI");
  }
  private native void afisareMesaj(String mesaj);
  static {
    System.loadLibrary("testAfisare");
  }
}
```

Metoda nativă `afisareMesaj()` va fi definită în biblioteca dinamică `testDoi.dll`.

Pasul 2. Compilarea programului de mai sus folosind comanda:

```
javac AfisareMesajJNI.java
```

care conduce la obţinerea fişierului `AfisareMesajJNI.class`.

Pasul 3. Crearea automată a fişierului interfaţă asociat fişierului `AfisareMesajJNI.class` folosind comanda:

```
javah -jni AfisareMesajJNI
```

Această comandă va genera pentru fiecare declaraţie de metodă `native` câte un prototip de funcţie într-un fişier antet, numit `AfisareMesajJNI.h`.

Pasul 4. Scrierea aplicaţiei C care va genera *biblioteca de legare dinamică* (engl. *Dynamic Link Library* – DLL). Aplicaţia de mai jos include fişierul de interfaţă `AfisareMesajJNI.h` şi implementează metoda nativă `afisareMesaj()`. Deoarece această parte este dependentă de platformă, am scris aplicaţia în C Win32 API. Dacă se utilizează altă platformă, pentru a afişa mesajul se înlocuieşte `<windows.h>` cu `<stdio.h>` şi apelul funcţiei `MessageBox()` cu `printf()`.

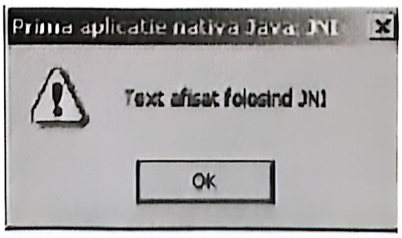
```
#include <windows.h>
#include "AfisareMesajJNI.h"

BOOL APIENTRY DllMain(HANDLE hModule,
  DWORD dwReason, void ** lpReserved) {
    return TRUE;
}

JNIEXPORT void JNICALL Java_AfisareMesajJNI_afisareMesaj
(JNIEnv * jEnv, jobject this, jstring jMsg) {
    const char * msg;
    msg = (*jEnv) -> GetStringUTFChars(jEnv, jMsg, 0);
    MessageBox(HWND_DESKTOP, msg,
      "Prima aplicatie nativa Java: JNI",
      MB_OK | MB_ICONEXCLAMATION);
    (*jEnv) -> ReleaseStringUTFChars(jEnv, jMsg, msg);
}
```

Folosind mediul Microsoft Visual C++ (de exemplu, versiunea 6.0), crearea unei biblioteci de legături dinamice se face după următorii paşi:

a) Se selectează (în ordine) `File`, `New`, apoi `Projects`;
b) Din lista de aplicaţii se selectează `Win32 DynamicLink Library`, iar apoi se completează numele proiectului (engl. *Project Name*) cu `testAfisare` (trebuie să coincidă cu numele argumentului metodei `loadLibrary()` din programul Java de la pasul 1);
c) Se adaugă la proiectul creat fişierul C de mai sus, după care se compilează întreg proiectul.

Astfel s-a creat în subdirectorul `Debug` fişierul `testAfisare.dll`.

Pasul 5. Execuţia aplicaţiei Java necesită existenţa bibliotecii `testAfisare.dll` în directorul în care se află fişierul `AfisareMesajJNI.class`. Lansând comanda `java AfisareMesajJNI`, obţinem o fereastră de dialog care conţine textul dorit:



Prima aplicatie nativa Java: JNI

Text afisat folosind JNI

Fereastra de dialog din Win 32 API

Continuăm cu prezentarea clauzei `throws` care apare în declaraţiile metodelor unei clase. Aceasta a mai fost prezentată succint în capitolul 2, secţiunea 2.9.11, exemplele 2.9.13, 2.9.14, 2.9.16, 2.9.18, când s-a descris instrucţiunea `throw`. O clauză `throws` este utilă la declararea oricăror excepţii ce pot rezulta în urma execuţiei unei metode sau a unui constructor. Aşa cum s-a menţionat şi în definiţia unei metode, sintaxa generală a definiţiei clauzei `throws` este:

```
throws <TipClasa1> [, . . ., <TipClasaN>]
```

unde `<TipClasa1>` , ..., `<TipClasaN>` sunt clase `Throwable` sau subclase ale clasei `Throwable`, în caz contrar se obţine o eroare la compilare.

Corpul metodei va fi prezentat în continuare. Acesta poate fi bloc de instrucţiuni care implementează metoda sau doar indicând lipsa implementării. Corpul unei metode este `;` dacă şi numai dacă metoda este declarată `abstract` sau `native`. Dacă metoda este declarată `abstract` sau `native` şi este definită printr-un bloc de instrucţiuni, atunci se obţine eroare la compilare. Şi, reciproc, dacă o metodă nu este declarată `abstract` sau `native` şi nu este definită, atunci se obţine eroare la compilare. Dacă o metodă este declarată `void`, atunci corpul său nu trebuie să conţină o instrucţiune `return` cu expresie, în caz contrar apare o eroare de compilare. O metodă care are tip returnat trebuie să încheie execuţia metodei printr-o instrucţiune `return` cu expresie sau printr-o instrucţiune `throw`.

Exemplul 3.2.37. Programul Java de mai jos scoate în evidenţă o situaţie când nu toate ramurile unei instrucţiuni `if` au asociată o instrucţiune (obligatorie) `return`:

```
class C1 {
  int f(int x) {
    if (x < 10) return x;
    else throw new RuntimeException("x este prea mare!"); }
}

public class TestReturn {
  public static void main(String args[]) {
    C1 obiect = new C1();
    System.out.println(obiect.f(9));
    System.out.println(obiect.f(12));
  }
}
```

Acesta va afişa la execuţie:

```
9
java.lang.RuntimeException:
x este prea mare!
        at C1.f(TestReturn.java:4)
        at TestReturn.main(TestReturn.java:11)
Exception in thread "main"
```

Continuăm cu noţiunile de suprascriere, ascundere şi supraîncărcare a metodelor unei clase. O clasă moşteneşte de la superclasa directă şi de la superinterfeţele directe toate metodele (chiar dacă sunt declarate `abstract`) accesibile din codul clasei şi care nu sunt suprascrise şi nu sunt ascunse de declaraţia clasei. Dacă o clasă declară o metodă instanţă, spunem că declaraţia metodei *suprascrie* toate metodele cu aceeaşi signatură din superclasele şi superinterfeţele clasei (care altfel ar fi fost accesibile codului clasei). În plus, dacă declaraţia metodei din clasă nu este `abstract`, atunci spunem că declaraţia metodei *implementează* toate metodele cu aceeaşi signatură din superclasele şi superinterfeţele clasei (care altfel ar fi fost accesibile codului clasei).

O metodă instanţă nu poate suprascrie o metodă declarată `static` (din acest punct de vedere, suprascrierea metodelor diferă de ascunderea atributelor, unde era permis pentru o variabilă instanţă să ascundă o variabilă `static`), în caz contrar apare o eroare la compilare.

O metodă suprascrisă poate fi accesată prin utilizarea unei expresii care conţine cuvântul rezervat `super`. Spre deosebire de accesul la un atribut ascuns, apelul unei metode suprascrise nu se poate face prin nume calificat sau printr-o conversie la tipul superclasei.

Exemplul 3.2.38. Programul Java de mai jos pune în evidenţă o situaţie de suprascriere a unei metode instanţă. Astfel, se va apela metoda `f()` a clasei `C1` la fiecare apel al lui `f()` al clasei `C2`, chiar dacă tipul obiectului curent este `C1`.

```
class C1 {
   int x = 1;
   void f(int x) {
      this.x = x;
      System.out.println("Metoda f() din C1");
   }
}

class C2 extends C1 {
   int xDoi = 5;
   void f(int x) {
      System.out.println("Metoda f() din C2");
      super.f(g(x));
      System.out.println("Metoda f() din C2");
   }
   int g(int x) {
      return x > xDoi ? x : xDoi;
   }
}

public class Suprascriere {
   public static void main(String args[]) {
      C1 obiect = new C2();
```

```
        obiect.f(4);
        System.out.println(obiect.x);
    }
}
```

Acest program va afişa la execuţie:

```
Metoda f() din C2
Metoda f() din C1
Metoda f() din C2
5
```

Dacă o clasă declară o metodă statică, atunci spunem că declaraţia metodei *ascunde* toate metodele cu aceeaşi signatură din superclasele şi superinterfeţele (care altfel ar fi fost accesibile codului clasei). Dacă o metodă statică ascunde o metodă instanţă, atunci apare o eroare la compilare (spre deosebire de ascunderea unui atribut care este permisă). O metodă ascunsă poate fi accesată prin nume calificat, printr-o expresie care conţine cuvântul rezervat `super` sau cu o conversie la un tip superclasă (la fel ca la ascunderea atributelor).

Dacă o declaraţie de metodă suprascrie sau ascunde declaraţia altei metode, atunci acestea trebuie să returneze acelaşi tip. Modificatorul de acces al unei metode suprascrise şi ascunse trebuie să furnizeze cel puţin la fel de mult acces ca al metodei suprascrise, în caz contrar apare o eroare de compilare, astfel:

- dacă metoda suprascrisă sau ascunsă este `public`, atunci metoda care o suprascrie sau o ascunde trebuie să fie declarată `public`;
- dacă metoda suprascrisă sau ascunsă este `protected`, atunci metoda care o suprascrie sau o ascunde trebuie să fie `public` sau `protected`;
- dacă metoda suprascrisă sau ascunsă are acces implicit (fără a preciza nici un modificator), atunci metoda care o suprascrie sau o ascunde nu trebuie să fie `private`.

O metodă declarată `private` nu este accesibilă subclaselor şi deci nu poate fi vorba despre suprascrierea sau ascunderea ei.

Exemplul 3.2.39. Programul de mai jos exemplifică ascunderea atributelor şi suprascrierea metodelor.

```
class C1 {
   int x = 1;
   void f(int x) { this.x = x; }
   int getX_C1() { return x; }
}

class C2 extends C1 {
   float x = 5.0f;
   void f(int x) { super.f((int)x); }
   float getX_C2() { return x; }
}

public class SuprascriereSiAscundere {
   public static void main(String args[]) {
      C2 obiect = new C2();
      obiect.f(4);
      System.out.println(obiect.getX_C2());
      System.out.println(obiect.getX_C1());
   }
}
```

Clasa `C2` ascunde declaraţia variabilei instanţă `x` a clasei `C1` cu variabila instanţă `x` a clasei `C1`, apoi suprascrie metoda `f()` a clasei `C1` cu propria metodă `f()`. La apelul `obiect.f(4)` se va modifica atributul `x` al clasei `C1` la valoarea `4`. Execuţia aplicaţiei va afişa:

```
5.0
4
```

Este posibil ca o clasă să moştenească mai mult de o metodă cu aceeaşi signatură, astfel:

1. Dacă una dintre metodele moştenite nu este abstractă, atunci există subcazurile:
   a) Dacă metoda care nu este abstractă este statică, atunci se obţine eroare la compilare;
   b) Altfel, metoda care nu este abstractă se consideră că suprascrie şi deci implementează toate celelalte metode pe care clasa le moşteneşte. Dacă tipul returnat al metodei care suprascrie este altul decât cel al metodelor suprascrise, atunci se obţine eroare la compilare. Acelaşi lucru se obţine dacă metoda care nu este abstractă are o clauză `throws`, care are conflicte cu altă metodă moştenită.
2. Dacă toate metodele moştenite sunt abstracte, atunci clasa este abstractă şi moşteneşte toate metodele abstracte. Metodele moştenite (suprascrise) trebuie să aibă acelaşi tip returnat.

Exemplul 3.2.40. Programul de mai jos pune în evidenţă faptul că la suprascrierea metodelor trebuie să coincidă şi lista excepţiilor clauzei `throws`, care pot fi aruncate în timpul apelului metodei:

```
class ExceptiaNoastra extends Exception {
   ExceptiaNoastra() { super(); }
}

class C1 {
   int x;
   void f(int x) { this.x += x; }
}

public class ClasaDerivata extends C1 {
  void f(int x) throws ExceptiaNoastra {
    if (x < 0) throw new ExceptiaNoastra();
    this.x = x;
  }
}
```

Programul va furniza eroare la compilare, deoarece metoda suprascrisă `f()` din `C1` nu are specificată nici o clauză `throws` aşa cum are metoda `f()` din `ClasaDerivata`. Evident, programul devine corect dacă adăugăm în antetul metodei `f()` din clasa `C1` clauza `throws ExceptiaNoastra`.

O metodă clasă ascunsă (statică) poate fi invocată folosind o referinţă al cărei tip este clasa care conţine declaraţia metodei. În concluzie, ascunderea metodelor statice diferă de suprascrierea metodelor instanţă.

Exemplul 3.2.41. Programul de mai jos evidenţiază diferenţa dintre ascunderea metodelor statice şi suprascrierea metodelor instanţă.

```
class C1 {
  static String f() { return "Mesajul Unu din C1"; }
  String g() { return "Mesajul Doi din C1"; }
}

class C2 extends C1 {
  static String f() { return "Mesajul Unu din C2"; }
  String g() { return "Mesajul Doi din C2"; }
}

public class DiferentaStaticInstanta {
  public static void main(String[] args) {
    C1 obiect = new C2();
    System.out.println(obiect.f() + ", " + obiect.g());
  }
}
```

Obiectul creat aparţine clasei `C1` şi deci apelul `obiect.f()` se va referi la metoda statică `f()` din clasa `C1`. În schimb, apelul `obiect.g()` se va referi la metoda `g()` din clasa `C2`, deoarece aceasta suprascrie metoda `g()` din clasa `C1`. Astfel, execuţia va afişa:

```
Mesajul Unu din C1, Mesajul Doi din C2
```

Continuăm cu supraîncărcarea metodelor unei clase. Dacă două metode ale unei clase (declarate în aceeaşi clasă sau moştenite de o clasă sau una declarată şi una moştenită) au acelaşi nume, dar signaturi diferite, atunci se spune că numele metodei este *supraîncărcat* (se mai spune că metoda este *supraîncărcată*). Metodele sunt suprascrise comparându-se signaturile. Dacă avem două metode declarate într-o clasă `C1`, să zicem `public void f();` şi `public void f(int);`, şi subclasa `C2` suprascrie una din ele, de exemplu `public void f();`, atunci subclasa `C2` va moşteni metoda cealaltă, adică `public void f(int);` (din acest punct de vedere, Java diferă de C++). În cazul metodelor supraîncărcate, numărul de parametri actuali şi tipurile argumentelor din momentul compilării vor determina signatura metodei care va fi apelată. Dacă metoda care trebuie apelată este metodă instanţă, metoda actuală care trebuie să fie apelată va fi determinată în momentul execuţiei, folosind metode dinamice.

Exemplul 3.2.42. Se evidenţiază supraîncărcarea metodelor unei clase.

```
class C1 {
  double x = 5.0f;
  void f(int x) {
    f((double)x + 2);
  }
  void f(double x) {
    this.x = x;
  }
}
public class Supraincarcare {
  public static void main(String args[]) {
    C1 obiect = new C1();
    obiect.f(4.0);
    System.out.println(obiect.x);
  }
}
```

Fiind metodă instanţă, la momentul execuţiei se va decide care metodă `f()` se va apela. Cum literalul `4.0` este de tip `double`, evident se va apela `f(double)` şi programul va afişa la execuţie:

```
4.0
```

O metodă poate avea un număr variabil de parametri, caz în care vor fi furnizaţi metodei sub formă de tablou. În mod obligatoriu, partea variabilă de parametri se află la sfârşitul listei de parametri formali ai metodei.

Exemplul 3.2.43. Aplicaţia de mai jos conţine două metode cu număr variabil de parametri:

```
public class ListaParametri {

  public static double aduna(double ... params){
    System.out.println("Numarul de parametri: " +
        params.length);
    double suma = 0;
    for (double valoare : params)
```

```
        suma += valoare;
      return suma;
    }

    public static double inmulteste(double a, double b,
          double ... ds) {
      double inmultire = a * b;
      for (double valoare : ds)
        inmultire *= valoare;
      return inmultire;
    }

    public static void main(String[] args) {
      System.out.println(aduna());
      System.out.println(aduna(3));
      System.out.println(aduna(2.1, 3.14));
      System.out.println(aduna(5, 3.5, -2));

      System.out.println(inmulteste(3.19, 2.27));
      System.out.println(inmulteste(2.02, 2.75, 12, 0.03));
    }

  }
```

Metoda `aduna()` poate avea zero sau mai mulţi parametri. Parametrul `params` este, de fapt, un tablou de elemente `double`. Metoda `inmulteste()` are cel puţin doi parametri de tip `double`. Nu este necesar ca toţi parametrii să fie de acelaşi tip; de exemplu, parametrul `a` poate fi de tip `float`.

În urma execuţiei programului de mai sus se va obţine:

```
Numarul de parametri: 0
0.0
Numarul de parametri: 1
3.0
Numarul de parametri: 2
5.24
Numarul de parametri: 3
6.5
7.2413
1.9997999999999998
```

3.2.7. Iniţializatori statici

Iniţializatorii statici declaraţi într-o clasă sunt executaţi când clasa este iniţializată. Împreună cu iniţializatorii atributelor pentru variabilele clasă pot fi utilizaţi pentru iniţializarea variabilelor clasă. Sintaxa generală a iniţializatorilor statici este:

```
static <Bloc>
```

Iniţializatorii statici şi iniţializatorii variabilelor clasă sunt executaţi în ordinea în care apar şi nu se pot referi la variabilele clasă declarate după utilizarea lor, chiar dacă aceste variabile de clasă sunt în domeniu. Această restricţie are ca scop determinarea la momentul compilării a iniţializărilor circulare. Într-un iniţializator static nu poate apărea instrucţiunea `return` sau cuvintele rezervate `this` şi `super`, în caz contrar semnalându-se o eroare la compilare.

Exemplul 3.2.44. Următorul program Java justifică importanţa ordinii iniţializatorilor statici:

```
class C1 {
  static int f() { return j; }
  static int i = f();
  static int j = 1;
}

public class TestStatic {
  public static void main(String[] args) {
    System.out.println(C1.i);
  }
}
```

Iniţializatorul variabilei `i` foloseşte metoda `f()` pentru a accesa valoarea variabilei `j` înainte ca `j` să fie iniţializat cu `1`. Astfel, execuţia programului de mai sus va afişa: `0`.

3.2.8. Declaraţiile constructorilor

Un *constructor* este utilizat la crearea unui obiect care reprezintă o instanţă a unei clase. Declaraţia unui constructor arată ca declaraţia unei metode, dar nu are tip rezultat (nu returnează nimic, nici măcar `void`):

```
[public | protected | private] <NumeClasa>
  ([<ListaParametriFormali>])
     [throws <ListaClase>] {
  [<ApelExplicitConstructor>]
     [<BlocInstructiuni>]
}
```

unde:

1. `<NumeClasa>` reprezintă numele clasei în care este declarat constructorul;
2. `<ListaParametriFormali>` are aceeaşi sintaxă şi semantică ca la declaraţia metodelor (astfel şi constructorii pot avea un număr fix sau variabil de parametri);
3. clauza opţională `throws <ListaClase>` are aceeaşi sintaxă şi semantică ca la declaraţia metodelor;
4. `<ApelExplicitConstructor>` poate fi de forma:
   a) `this(<ListaArgumente>);`
   b) `super(<ListaArgumente>);`

Constructorii sunt apelaţi de expresii de creare a instanţei clasei, de apelul metodei `newInstance()` a clasei `Class`, de conversiile şi concatenările rezultate în urma operatorului de concatenare + şi de apelul explicit al acestora din alţi constructori. Constructorii nu pot fi apelaţi în expresii de apel al unei metode. Accesul la constructori se realizează după modificatorii de acces. Constructorii nu sunt membri, deci nu pot fi niciodată subiect de ascundere sau suprascriere. Signatura unui constructor este similară cu structura şi comportarea signaturii unei metode.

Spre deosebire de metode, un constructor nu poate fi `final`, `abstract`, `static`, `synchronized` sau `native`. Un constructor nu se moşteneşte, deci nu are sens să fie declarat `final` sau `abstract`. Un constructor se apelează relativ la un obiect, deci nu are sens să fie `static`. Nu există o necesitate practică pentru ca un constructor să fie `synchronized`, deoarece acesta va bloca obiectul care se construieşte, ce oricum nu este accesibil altor fire de execuţie.

O clasă poate interzice crearea de instanţe ale clasei prin declararea a cel puţin unui constructor (pentru a preveni crearea unui constructor implicit) şi declarând toţi constructorii `private`.

Exemplul 3.2.45. Următorul program Java pune în evidenţă situaţia când o clasă nu poate avea instanţe:

```
class C1 {
  int x = 1;
  private C1() {}
}
```

Clasa `C1` nu poate avea instanţe; încercarea de a crea obiecte din clasa `C1` conduce la o eroare la compilare. Putem permite crearea de obiecte doar în acelaşi pachet prin acces implicit la constructor (fără modificatori), în rest nefiind posibilă crearea de obiecte:

```
package P1;

public class C1 {
  int x = 1;
  C1() {}
}
```

Exemplul 3.2.46. Următorul program Java continuă exemplul 3.2.41, în sensul „controlării" mai exacte a numărului de instanţe pe care le poate avea o clasă:

```
class ExceptiaNoastra extends Exception {
  ExceptiaNoastra(String s) { super(s); }
}

class C1 {
  static int numarInstante = 0;
  static int numarMaximInstante = 3;
  int x = 1;
  C1(int x) throws ExceptiaNoastra {
    numarInstante ++;
    if (numarInstante > numarMaximInstante)
      throw new ExceptiaNoastra(
        "Am depasit numarul de instante");
    System.out.println("Obiect " + numarInstante);
    this.x = x;
  }
}

public class NumarMaximDeInstante {
  public static void main(String args[]) {
    try {
      C1 obiectUnu = new C1(1);
      C1 obiectDoi = new C1(2);
      C1 obiectTrei = new C1(3);
      C1 obiectPatru = new C1(4);
    }
    catch (ExceptiaNoastra e) {
      System.out.println("Exceptie raportata: " + e);
    }
  }
}
```

Execuţia acestui program conduce la afişarea:

```
Obiect 1
Obiect 2
Obiect 3
Exceptie raportata: ExceptiaNoastra: Am depasit numarul
de instante
```

Prima instrucţiune a corpului unui constructor poate fi apelul explicit al altui constructor al aceleiaşi clase cu ajutorul cuvântului rezervat `this`, urmat de o listă de argumente, sau apelul explicit al unui constructor al unei superclase directe, utilizând cuvântul rezervat `super`, urmat de o listă de argumente între paranteze. Apelul explicit al unui alt constructor folosind `this` sau `super` trebuie să fie prima instrucţiune, altfel se obţine eroare la compilare. Astfel, apelarea directă sau indirectă a unui constructor cel puţin o dată conduce la eroare de compilare. În cazul în care corpul unui constructor nu începe cu apelul explicit al altui constructor şi constructorul declarat nu este parte a clasei `Object`, atunci se presupune implicit de către compilator că se începe corpul constructorului cu un apel al constructorului superclasei directe fără argumente (adică de forma `super();`). Cu excepţia apelurilor explicite ale constructorilor, corpul unui constructor este similar corpului unei metode, putând conţine chiar instrucţiuni `return`, însă fără expresie ataşată. Variabilele instanţă (adică atributele nestatice), metodele instanţă declarate într-o clasă sau într-o superclasă, cuvintele rezervate `this` şi `super` nu pot fi folosite în apelul unui constructor explicit, în caz contrar apare o eroare la compilare.

Exemplul 3.2.47. Următorul program Java evidenţiază apelurile explicite ale constructorilor prin intermediul cuvintelor rezervate `this` şi `super`:

```
class C1 {
   int x = 1;
   C1(int x) { this.x = x; }
}

class C2 extends C1 {
   static final int y = 2;
   int z = 3;
   C2(int x) {
      this(x, y);
   }
   C2(int x, int z) {
      super(x);
      this.z = z;
   }
   public String toString() {
      return "x = " + x + ", y = " + y + ", z = " + z;
   }
}

public class ThisSuper {
   public static void main(String args[]) {
      C2 obiectUnu = new C2(4);
      C2 obiectDoi = new C2(5, 6);
      System.out.println(obiectUnu);
      System.out.println(obiectDoi);
   }
}
```

Primul constructor al clasei `C2` apelează cel de-al doilea constructor adăugând ca argument atributul `y`. Al doilea constructor al clasei `C2` apelează constructorul clasei

`C1`, iniţializând atributul `x`. Dacă primul constructor al clasei `C2` ar fi avut un apel explicit de forma `this(x, z)`, atunci am fi obţinut eroare la compilare, deoarece în apelul explicit al unui constructor nu putem avea variabile instanţă. Programul va afişa la execuţie:

```
x = 4, y = 2, z = 2
x = 5, y = 2, z = 6
```

Supraîncărcarea constructorilor este similară cu cea a metodelor. Supraîncărcarea este rezolvată în timpul compilării pentru fiecare expresie de creare a instanţei clasei.

Dacă o clasă nu conţine nici o declaraţie de constructor, atunci compilatorul creează un *constructor implicit* fără parametri care apelează constructorul superclasei fără argumente.

Se obţine eroare la compilare în cazul în care compilatorul creează un constructor implicit şi superclasa acesteia are constructori, însă nici unul fără parametri.

Exemplul 3.2.48. Următoarele clase Java scot în evidenţă apelul constructorului implicit pentru clasa `C2`.

```
class C1 {
  int x = 1;
  C1(int x) { this.x = x; }
}

class C2 extends C1 {
  int y = 2;
}
```

Acestea vor conduce la eroare de compilare, deoarece constructorul implicit al clasei `C2` va apela constructorul implicit al clasei `C1` care nu există, deoarece a fost definit doar un singur constructor cu parametru.

Dacă o clasă este declarată `public`, atunci constructorul implicit are acces `public`; altfel, în cazul absenţei modificatorului de acces, constructorul implicit are acces implicit (engl. *friendly*).

Exemplul 3.2.49. Următoarea declaraţie de clasă Java:

```
public class C {
  int x;
}
```

este echivalentă cu declaraţia:

```
public class C {
  int x;
  public C() { super(); }
}
```

Continuăm cu prezentarea relaţiei dintre constructorii clasei derivate şi ai superclasei acesteia. Când se creează un obiect al clasei derivate, acesta conţine un subobiect al clasei de bază. Acest subobiect al clasei de bază trebuie iniţializat corect, adică prin apelul explicit al constructorului clasei de bază (care poate iniţializa subobiectul clasei de bază). În cazul constructorilor impliciţi (fără parametri), Java inserează apelul constructorului implicit al clasei de bază înaintea apelului efectiv al constructorului clasei curente. În cazul constructorilor cu parametri, trebuie făcut apelul constructorului clasei de bază în mod explicit (utilizând cuvântul rezervat `super`), altfel se obţine eroare la compilare.

Exemplul 3.2.50. Următorul program Java evidenţiază ordinea apelurilor constructorilor impliciţi pentru clasele derivate.

```
class C1 {
  C1() { System.out.println("Constructorul clasei C1"); }
}
class C2 extends C1 {
  C2() { System.out.println("Constructorul clasei C2"); }
}
public class OrdineConstructori extends C2 {
  OrdineConstructori() {
    System.out.println(
      "Constructorul clasei OrdineConstructori");
  }
  public static void main(String[] args) {
    OrdineConstructori x = new OrdineConstructori();
  }
}
```

Clasa `OrdineConstructori` este derivată din `C2`, care la rândul ei este derivată din `C1`. Astfel, se va apela întâi constructorul superclasei `C1`, apoi al clasei `C2` şi în final cel al clasei `OrdineConstructori`. Aşadar, programul va afişa la execuţie:

```
Constructorul clasei C1
Constructorul clasei C2
Constructorul clasei OrdineConstructori
```

Exemplul 3.2.51. Următorul program Java scoate în evidenţă apelurile constructorilor cu parametri pentru clasele derivate. Spre deosebire de exemplul 3.2.47, clasa derivată va trebui să conţină ca primă instrucţiune în constructorul său un apel al constructorului explicit al clasei derivate folosind `super`; în caz contrar se obţine eroare la compilare.

```
class C1 {
  int x = 1;
  C1 (int x) { this.x = x; }
}
class C2 extends C1 {
  int x = 2, y = 3;
  C2(int x) { super(x); }
  C2(int x, int y) { super(x); this.y = y; }
}
public class ApelExplicitConstructor {
  public static void main(String args[]) {
    C1 obiectUnu = new C1(4);
    C2 obiectDoi = new C2(5);
    C2 obiectTrei = new C2(6, 7);
    System.out.print(obiectUnu.x + " ");
    System.out.print(obiectDoi.x + " " + obiectDoi.y + " ");
    System.out.print(obiectTrei.x + " " + obiectTrei.y);
  }
}
```

Dacă în declaraţiile constructorilor clasei `C2` ar fi lipsit instrucţiunea `super(x)`, atunci s-ar fi obţinut eroare la compilare. La crearea unui obiect din `C2`, întâi se apelează constructorul din clasa `C1`, apoi cel din clasa `C2`. Rezultă imediat că pentru toate obiectele din `C2`, atributul `x` va fi întotdeauna egal cu `2` (la apelul `super(x)` este asignat întâi `1` şi în cele din urmă `2`). Astfel, programul va afişa la execuţie:

```
4 2 3 2 7
```

3.2.9. Clase interioare

O *clasă interioară* (engl. *inner classes*) este o clasă definită în interiorul altei clase. Accesul la membrii clasei interioare se face analog celui la membrii claselor obişnuite. De exemplu, dacă o clasă interioară este calificată `private`, atunci putem să o folosim doar în clasa exterioară unde este definită. Clasele interioare sunt importante pentru că permit gruparea claselor logic asemănătoare fără a coincide cu compunerea claselor. Clasele interioare sunt folosite frecvent în construirea obiectelor *listener* pentru rezolvarea evenimentelor (vezi capitolul despre Swing).

Exemplul 3.2.52. Următorul program Java pune în evidenţă accesul la o dată `private` dintr-o clasă interioară:

```
class Exterior {
  private int x = 2;
  public class Interior {
     private int y = 5;
     public void metodaInterioara() {
       System.out.println("x = " + x);
       x = 3;
```

```
            System.out.println("x = " + x);
            System.out.println("y = " + y);
        }
    } // sfarsitul clasei Interior
    public void metodaExterioara() {
     System.out.println("x = " + x);
    }
}
```

La compilarea acestui program, se vor forma două clase: `Exterior` şi `Exterior.Interior`. Codurile binare Java (*byte code*) corespunzătoare generate pe disc vor fi `Exterior.class` şi `Exterior$Interior.class` (se utilizează caracterul „`$`" în loc de „`.`" pentru a nu fi confundat cuvântul `Interior` cu o extensie a unui program). Când este creată o instanţă a clasei interioare, trebuie să existe de obicei o instanţă a clasei exterioare. Această instanţă a clasei exterioare este accesibilă din obiectul interior. Astfel, în exemplul precedent, variabilele `x` şi `y` sunt accesibile din clasa `Interior`. Programul va afişa la execuţia metodei `metodaInterioara()`:

```
x = 2
x = 3
y = 5
```

Accesibilitatea membrilor clasei exterioare este posibilă, deoarece clasa interioară are, de fapt, o referinţă ascunsă către instanţa clasei exterioare care a fost contextul curent când clasa interioară a fost creată. De fapt, aceasta asigură că şi clasa interioară, şi cea exterioară aparţin împreună, şi nu faptul că instanţa interioară este doar un membru al instanţei exterioare.

Exemplul 3.2.53. Uneori dorim să creăm o instanţă a unei clase interioare dintr-o metodă statică sau când nu avem la dispoziţie un obiect `this` disponibil. Iată cum putem defini un obiect dintr-o clasă interioară fără să utilizăm cuvântul rezervat `this`:

```
public static void main(String args[]) {
  Exterior.Interior obiect = new Exterior().new Interior();
  obiect.metodaInterioara();
}
```

Acesta poate fi „decompactat" în mod echivalent:

```
public static void main(String args[]) {
  Exterior obiectTemporar = new Exterior();
  Exterior.Interior obiect = obiectTemporar.new Interior();
  obiect.metodaInterioara();
}
```

Putem defini şi *clase interioare statice*. Acestea nu mai au referinţa către clasa exterioară. Deci metodele unei clase interioare statice nu pot accesa variabile instanţă ale clasei exterioare.

Putem defini, de asemenea, clase în interiorul unei metode din altă clasă (engl. *local nested class*). Putem crea obiecte ale unei clase interioare locale doar în metoda în care apare definiția clasei. Aceasta este utilă atunci când calculul într-o metodă necesită folosirea unei clase specializate care nu este cerută sau folosită în altă parte. În plus față de clasele definite în interiorul claselor, avem două aspecte particulare:

1. Un obiect creat într-o clasă interioară unei metode are „alt" acces la variabilele metodei exterioare;
2. Este posibil să creăm o clasă anonimă (adică o clasă fără nume specificat).

Descriem pe rând cele două aspecte particulare prezentate mai sus:

1. Orice variabilă, locală sau parametru formal, poate fi accesată de clasa interioară numai dacă este calificată `final` (adică este o constantă). Permițând accesul numai la variabilele `final`, este posibil să copiem valorile acestor variabile chiar în obiect, extinzând astfel viața obiectelor clasei interioare.

Exemplul 3.2.54. Următorul program Java pune în evidență apelul unei metode a unei clase interne altei clase.

```
public class ClasaInterioara {
  public static void main(String args[]) {
    Exterior obiectExterior = new Exterior();
    // apelam o metoda a unei clase interne unei alte clase
    Exterior.Interior obiectInterior = obiectExterior.new
    Interior();
    obiectInterior.metodaInterioara();
    // apelam o metoda a unei clase care construieste un obiect
    // al altei clase
    obiectExterior.f(4, 3);
  }
}
class Exterior {
  private int x = 2;
  public class Interior {
    private int y = 5;
    public void metodaInterioara() {
      System.out.println("x = " + x);
      System.out.println("y = " + y);
    }
  }
  public void metodaExterioara() {
    System.out.println("x = " + x);
  }
  public void f(int xx, final int yy) {
    int a = xx + yy;
    final int b = xx - yy;
```

```
      class Interioara {
        public void metodaInterioara() {
          System.out.println("x = " + x);
          System.out.println("b = " + b);
        }
      }
      Interioara obiectInteriorSecund = new Interioara();
      obiectInteriorSecund.metodaInterioara();
    }
  }
```

`obiectInterior` este un obiect al clasei `Exterior.Interior` şi deci apelul lui `metodaInterioara()` va implica afişarea atributelor private `x` şi `y` (accesibile clasei interioare). Apelarea metodei `f()` asupra lui `obiectExterior` va implica crearea lui `obiectInteriorSecund`, căruia i se aplică `metodaInterioara()`. Aceasta va afişa valorile atributului final `b` şi ale atributului `x` al clasei `Exterior`. Nu putem accesa variabila locală a metodei `f()` dintr-o clasă interioară, în caz contrar se obţine eroare la compilare. Astfel, programul va afişa la execuţie:

```
x = 2
y = 5
x = 2
b = 1
```

2. Pentru clase anonime, nu putem utiliza operatorul `new` în modul bine cunoscut. De fapt, clasele anonime sunt definite în locul unde sunt construite.

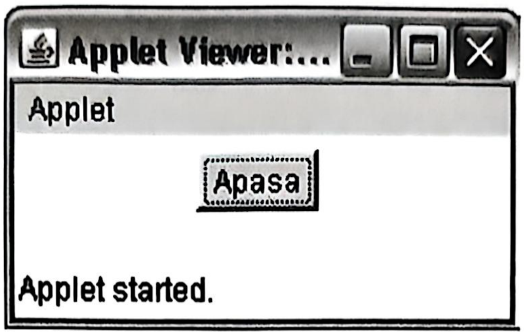
Exemplul 3.2.55. Programul următor reprezintă un applet în care utilizăm clase anonime:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Buton extends Applet {
  private Button buton;
  public void init() {
    buton = new Button("Apasa-ma");
    add(buton);
    buton.addActionListener(new ActionListener() {
       public void actionPerformed(ActionEvent e) {
         System.out.println("A aparut o actiune");
       }
    });
  } // sfarsitul definitiei metodei init()
} // sfarsitul definitiei clasei Buton
```

Apelul lui `new` este urmat de definiţia clasei (anonime). Clasa nu are nume, dar este referită printr-un nume de interfaţă (fără să folosească cuvântul rezervat `implements`). Nu putem instanţia clase anonime decât în modul de mai sus. Este recomandat ca şi clasele anonime să fie mici. O clasă anonimă poate fi o subclasă a unei clase explicite sau poate implementa o singură interfaţă explicită. Dacă o clasă anonimă extinde o clasă existentă, atunci argumentele constructorului superclasei trebuie puse în partea de argumente a expresiei `new`. De exemplu, putem avea în programul de mai sus expresia:

```
new Button("Apasa") { // modificari pentru Button ... }
```

Execuţia appletului prezentat conduce la afişarea:



Fereastra appletului care foloseşte o clasă anonimă

3.2.10. Clase generice

Există situaţii în care nu ştim cu ce tipuri de date vom lucra. Acest lucru va fi ştiut în momentul execuţiei sau în anumite cazuri particulare de utilizare.

De exemplu, putem lucra cu mulţimi de obiecte. Acestea pot avea tipuri diferite sau, cum se întâmplă în majoritatea cazurilor, acelaşi tip. Ar fi costisitor să definim câte o clasă pentru implementarea unei mulţimi şi a operaţilor aferente pentru fiecare tip de dată în parte. Singurul lucru care ar diferi (în afara numelui de clasă) ar fi tipul obiectelor din mulţime. Pentru rezolvarea acestui neajuns, Java oferă posibilitatea de a defini clase generice.

O *clasă generică* este o clasă care utilizează tipuri generice, acestea nefiind specificate în momentul definirii clasei. Tipul generic poate fi interpretat ca o variabilă de tip. În momentul utilizării clasei se poate stabili care sunt tipurile de date ce sunt utilizate efectiv.

Ca tipuri generice se pot utiliza doar tipurile referinţă. În cazul tipurilor primitive se pot utiliza clasele *wrapper* asociate. Se poate lucra cu valorile primitive, conversia la tipurile *wrapper* realizându-se implicit.

Exemplul 3.2.56. O clasă generică este implementată mai jos. Aceasta defineşte o pereche de atribute de acelaşi tip.

```
public class Pereche<T> {
  T p1, p2;
  Pereche() {
  }

  Pereche(T p1, T p2) {
    this.p1 = p1;
    this.p2 = p2;
  }
}
```

Tipul `T` va fi stabilit la utilizarea clasei `Pereche`. Clasa de mai jos sugerează câteva modalităţi de utilizare a clasei generice `Pereche`.

```
public class PerecheTest {
  public static void main(String[] args) {
    Pereche<String> pereche1 = new Pereche<String>();
    pereche1.p1 = "Primul element";
    pereche1.p2 = "Al doilea element";

    Pereche<Integer> pereche2 = new Pereche<Integer>(1, 2);
    System.out.println(pereche2.p1.doubleValue());
    System.out.println(pereche2.p1 + pereche2.p2);

    Pereche<Double> pereche3 = new Pereche<Double>();
    pereche3.p1 = 1d;
    pereche3.p2 = 2.0;
    System.out.println(pereche3.p1 + pereche3.p2);
    System.out.println(pereche3.p1 + pereche2.p2);

    Pereche pereche4 = new Pereche();
    pereche4.p1 = 5d;
    pereche4.p2 = 1;
  }
}
```

Pentru prima pereche, `T` va fi înlocuit cu tipul `String`. La a doua pereche se utilizează tipul `Integer` şi se apelează la constructorul cu parametri ai clasei. În acest caz parametrii sunt de tip `int`, dar se realizează o conversie implicită la tipul `Integer`. În expresia `pereche2.p1 + pereche2.p2`, operanzii sunt de tip `Integer` şi sunt convertiţi automat la tipul `int`, iar apoi are loc efectuarea operaţiei. Deci expresia se va evalua la valoarea `3`. La perechea a treia trebuie neapărat să indicăm valori de tip `double` sau obiecte `Double`. De aceea sunt necesare utilizarea sufixului `d` pentru valorile numerice, scrierea numerelor folosind simbolul zecimal sau întrebuinţarea operatorului de conversie explicită. Altfel s-ar obţine o eroare la compilare. A patra pereche evidenţiază un caz în care sunt utilizate tipuri distincte: `Double`, respectiv `Integer`. Însă este descurajată utilizarea clasei fără a se stabili tipul (tipurile) generice.

Pentru generalitate se poate utiliza clasa `Object`:

```
    Pereche<Object> pereche5 = new Pereche<Object>();
    pereche5.p1 = 12;
    pereche5.p2 = "Zet";
```

Exemplul 3.2.57. O clasă generică poate avea mai multe tipuri generice. Clasa de mai jos utilizează trei tipuri generice.

```
class Tripleta<T1, T2, T3> {
  T1 t1;
  T2 t2;
  T3 t3;

  public Tripleta() {
  }

  public Tripleta(T1 t1, T2 t2, T3 t3) {
    this.t1 = t1;
    this.t2 = t2;
    this.t3 = t3;
  }

  public String toString() {
    return
      t1.getClass().getName() + " - " +
      t2.getClass().getName() + " - " +
      t3.getClass().getName() + "\n" +
      t1.toString() + " - " +
      t2.toString() + " - " +
      t3.toString();
  }
}
```

Câteva modalități de utilizare a clasei `Tripleta` sunt cuprinse în programul următor:

```
public class TripletaTest {
  public static void main(String[] args) {
    Tripleta tripleta1 =
      new Tripleta<Integer, String, Double>(-2, "qiu", 5.1);
    System.out.println(tripleta1);

    Tripleta<String, String, Double> tripleta2 =
      new Tripleta<String, String, Double>();
    tripleta2.t1 = "alfa";
    tripleta2.t2 = "qiu";
```

```
        tripleta2.t3 = 5.1;
        System.out.println(tripleta2);
    }
}
```

Pentru prima tripletă nu s-au stabilit în declararea variabilei care sunt tipurile generice. De aceea atributele `t1`, `t2` şi `t3` sunt considerate ca fiind de tip `Object`. Pentru a le utiliza ca `Integer`, `String` şi respectiv `Double` trebuie să folosim conversii explicite la aceste tipuri.

Pentru cea de-a doua tripletă s-a stabilit la declararea variabilei care sunt tipurile generice, caz în care `t1` şi `t2` sunt recunoscute implicit ca fiind de tip `String`, iar `t3` ca fiind `Double`.

Aplicaţia va afişa la consolă:

```
java.lang.Integer - java.lang.String - java.lang.Double
-2 - qiu - 5.1
java.lang.String - java.lang.String - java.lang.Double
alfa - qiu - 5.1
```

3.2.11. Enumerări

Sunt cazuri în care pentru anumite atribute putem avea doar un set finit de valori – de exemplu, pentru numărul lunii, pentru ziua din săptămână sau pentru anotimp. De regulă, în aceste situaţii, definim o serie de valori constante de tip numeric, de exemplu `int`. Însă această soluţie nu ne asigură că atributul respectiv nu poate lua şi alte valori în afară de constantele definite.

Pentru a preveni acest lucru, Java pune la dispoziţie enumerările. Astfel putem defini doar un set de valori care pot fi luate de un atribut la un moment dat.

Definim valorile pentru o serie de fructe:

```
public enum Fructe { MERE, PERE, PRUNE, CAISE, PIERSICI}
```

Declaraţia de mai sus conduce la generarea clasei `Fructe` care implementează interfeţele `Comparable` şi `Serializable`. Clasa generată are:

- atributele statice `MERE`, `PERE`, `PRUNE`, `CAISE`, `PIERSICI` de tip `Fructe`;
- metoda `public static Fructe[] values()` care furnizează valorile posibile pentru tipul `Fructe`;
- metoda `public static Fructe valueOf(String string)` care returnează obiectul de tip `Fructe` corespunzător şirului de caractere specificat;
- suprascrie metodele `equals()`, `hashCode()`, `toString()` şi `compareTo`.

Constructorii clasei nu sunt accesibili (fiind declaraţi `private`) astfel că nu se pot defini alte instanţe pentru clasa `Fructe`. Se pot utiliza doar cele specificate în definiţia enumerării, adică cele date de atributele statice ale clasei.

Exemplul 3.2.58. Aplicaţia următoare utilizează enumeraţia de fructe definită mai sus:

```
public class FructeTeste {
    public static void main(String[] args){
      Fructe piersici = Fructe.PIERSICI;
      System.out.println(
         "Ieri am fost la piata si am cumparat " +
         piersici);

      System.out.println("Fructe recunoscute: ");
      for(Fructe fructe : Fructe.values()) {
        System.out.println(" * " + fructe);
      }
   }
}
```

În urma execuţiei se va afişa:

```
Ieri am fost la piata si am cumparat PIERSICI
Fructe recunoscute:
 * MERE
 * PERE
 * PRUNE
 * CAISE
 * PIERSICI
```

Pentru o clasă enumerare se pot defini metode şi atribute suplimentare. De exemplu, pentru enumerarea `Fructe` putem adăuga o metodă de listare a valorilor posibile.

```
public enum Fructe {
  MERE, PERE, PRUNE, CAISE, PIERSICI;

  public static void list() {
    System.out.println("Fructe recunoscute: ");
    for(Fructe fructe : Fructe.values()) {
      System.out.println(" * " + fructe);
    }
  }
}
```

Exemplul 3.2.59. Definim enumerarea `CifraRomana` pentru lucrul cu cifre romane. Pentru fiecare cifră romană vom asocia o valoare numerică cu echivalentul acesteia. Creăm şi o metodă pentru evaluarea numerelor scrise cu cifre romane.

```
public enum CifraRomana {
  I(1), V(5), X(10), L(50), C(100), D(500), M(1000);

  private int valoare;

  private CifraRomana(int valoare) {
      this.valoare = valoare;
    }

    public int valoare() {
      return valoare;
    }

    public static int valoare(String nrRoman) {
      int valoare = 0;
      CifraRomana cifra;
      CifraRomana temp = CifraRomana.I;
      for (int i=nrRoman.length()-1; i>=0; i--) {
        cifra = CifraRomana.valueOf(nrRoman.substring(i, i+1));
        if (cifra.valoare() < temp.valoare())
          valoare -= cifra.valoare();
        else {
          valoare += cifra.valoare();
          temp = cifra;
        }
      }
      return valoare;
    }
}
```

Constructorul clasei nu poate fi declarat `public` pentru că în acest caz s-ar putea obţine şi alte valori decât cele definite de enumerare. Orice încercare de definire a unui constructor public va conduce la apariţia unei erori la compilare. La declararea atributului `I(1)` se indică, de fapt, că se va apela constructorul cu un parametru de tip `int`.

Următorul program:

```
public class CifreRomaneTest {
  public static void main(String[] args) {
    System.out.println(
      "XIX = " + CifraRomana.valoare("XIX"));
    System.out.println(
      "DXIV = " + CifraRomana.valoare("DXIV"));
    System.out.println(
      "MMDCCXXXI = " + CifraRomana.valoare("MMDCCXXXI"));
  }
}
```

va afişa la execuţie:

```
XIX = 19
DXIV = 514
MMDCCXXXI = 2731
```

Exemplul 3.2.60. În exemplul precedent, metoda statică `valoare` este improprie pentru clasa `CifraRomana`. Ar trebui să avem doar metode pentru cifre romane, nu și pentru numere. De aceea, eliminăm această metodă și creăm clasa `NumarRoman`:

```
public final class NumarRoman {
  private CifraRomana[] numarRoman;

    /** Constructor fara parametri */
    public NumarRoman() {
      numarRoman = new CifraRomana[0];
    }

    /** Constructor cu un parametru de tip String,
      * care va contine un numar roman */
    public NumarRoman(String string) {
      setNumar(string);
    }

    /** Stabileste numarul roman tinut de obiectul curent */
    public void setNumar(String string) {
      numarRoman = new CifraRomana[string.length()];
      for(int i = 0; i<string.length(); i++)
        numarRoman[i] = CifraRomana.valueOf(string.substring
            (i, i+1));
    }

    /** Converteste numarul roman la String. */
    public String toString() {
      StringBuffer sb = new StringBuffer("");
      for (CifraRomana cifraRomana : numarRoman)
        sb.append(cifraRomana);
      return sb.toString();
    }

    /** Obtine valoarea numerica pentru numarul roman.*/
    public int valoare() {
      int valoare = 0;
      CifraRomana cifra;
      CifraRomana temp = CifraRomana.I;
      for (int i=numarRoman.length-1; i>=0; i--) {
         cifra = numarRoman[i];
         if (cifra.compareTo(temp) < 0)
           valoare -= cifra.valoare();
         else {
           valoare += cifra.valoare();
```

```
                temp = cifra;
            }
        }
        return valoare;
    }
}
```

Putem să rescriem programul de test de mai sus pentru a utiliza clasa `NumarRoman`:

```
public class CifreRomaneTest {
  public static void main(String[] args) {
    NumarRoman nr = new NumarRoman("XIX");
    System.out.println(nr + " = " + nr.valoare());
    nr = new NumarRoman("DXIV");
    System.out.println(nr + " = " + nr.valoare());
    nr.setNumar("MMDCCXXXI");
    System.out.println(nr + " = " + nr.valoare());
  }
}
```

Dacă în numărul roman se strecoară un caracter incorect, atunci se obţine la execuţie pentru metoda `valueOf()` excepţia `java.lang.IllegalArgumentException`.

3.2.12. Erori şi excepţii

Erorile şi excepţiile reprezintă situaţii particulare şi anormale care pot apărea în timpul execuţiei unei aplicaţii Java. Câteva cazuri de acest gen pot fi: eşuarea unei operaţii de intrare/ieşire, fişier inexistent, imposibilitatea de a face conversie de la un tip la altul, imprimare eşuată.

Superclasa tuturor excepţiilor şi a erorilor este `java.lang.Throwable`. Doar instanţele acestei clase (sau ale subclaselor acesteia) pot fi aruncate de maşina virtuală Java, sau pot apărea în cadrul instrucţiunii `throw`, sau într-o clauză `catch`, sau `throws`.

Clasa `Throwable` are două clase directe: `Error` pentru erori şi `Exception` pentru excepţii.

Erorile sunt situaţii grave care pot apărea şi, de regulă, execuţia programului nu poate continua sau anumite componente nu pot fi utilizate. Un exemplu de eroare este `VirtualMachineError` care poate apărea când maşina virtuală Java s-a defectat sau nu are suficiente resurse pentru funcţionare. Un alt exemplu de eroare este `IOError` care apare în timpul unor operaţii de intrare/ieşire.

Excepţiile sunt situaţii care pot apărea mai frecvent şi, de regulă, problemele apărute se pot remedia. De cele mai multe ori, aplicaţiile tratează aceste situaţii particulare.

Revenim la clasa `Throwable`. Aceasta are patru constructori:

```
public Throwable()
public Throwable(String message)
public Throwable(String message, Throwable cause)
public Throwable(Throwable cause)
```

Parametrul `message` stabileşte un mesaj care să descrie succint problema survenită, iar `cause` reprezintă situaţia de excepţie care a condus la apariţia celei curente.

Clasele `Error` şi `Exception` au tot patru constructori similari cu cei din clasa `Throwable`, având aceiaşi parametri şi aceeaşi semantică.

În tabelul de mai jos sunt prezentate cele mai utilizate metode publice definite în clasa `Throwable`.

Prototipul metodei	Descrierea metodei
`public Throwable` **`getCause`**`()`	Obţine eroarea sau excepţia care a condus la apariţia obiectului curent. Dacă eroarea sau excepţia curentă nu a fost declanşată de alta, se va obţine `null`.
`String` **`getMessage`**`()`	Obţine mesajul explicativ al problemei apărute.
`Throwable` **`initCause`**`(Throwable cause)`	Setează obiectul care a declanşat apariţia obiectului curent. Se poate utiliza şi valoarea `null` pentru parametru, caz în care se consideră că nu a fost o altă excepţie sau eroare care a condus la apariţia prezentei.
`void` **`printStackTrace`**`()`	Afişează la ieşirea standard de eroare lista metodelor apelate în momentul apariţiei excepţiei sau erorii curente.
`void` **`printStackTrace`**`(PrintStream s)`	Similar ca `printStackTrace()`, cu deosebirea că se va scrie într-un flux de tip `PrintStream`.
`void` **`printStackTrace`**`(PrintWriter s)`	Similar ca `printStackTrace()`, cu deosebirea că se va scrie într-un flux de tip `PrintWriter`.

O excepţie mai specială este `RuntimeException`, care apare în cazuri anormale întâlnite la momentul execuţiei. Excepţiile care au ca superclasă `RuntimeException`, excepţia `RuntimeException` sau orice eroare, nu sunt obligatorii a apărea în clauzele `throws` sau `catch`. În schimb, pentru excepţiile care nu pot fi convertite la `RuntimeException`, dacă apar într-o instrucţine `throw`, atunci instrucţiunea trebuie să fie cuprinsă într-un bloc `try` care să aibă o clauză `catch` cu care să se unifice. Aceeaşi situaţie este întâlnită când se apelează o metodă ce conţine în clauza `throws` o excepţie care nu este de clasă `RuntimeException` sau de o subclasă a sa.

Exemplul 3.2.61. În programul de mai jos sunt definite două excepţii utilizate în clasa `NumarPozitiv`.

```
class ExceptieUnu extends Exception {
  public ExceptieUnu() {
      super("Exceptia Unu");
    }
}

class ExceptieDoi extends RuntimeException {
    public ExceptieDoi() {
      super("Exceptia Doi");
    }
}

public class NumarPozitiv {

    int valoare;

    public NumarPozitiv(int valoare) throws ExceptieUnu {
      if (valoare < 0) throw new ExceptieUnu();
      this.valoare = valoare;
    }

    public NumarPozitiv scadere(NumarPozitiv np) {
      int rezultat = valoare - np.valoare;
      if (rezultat < 0) throw new ExceptieDoi();
      try {
        return new NumarPozitiv(rezultat);
      } catch(ExceptieUnu e) {
        throw new Error("Eroare grava!");
      }
    }

    public String toString() {
      return "" + valoare;
    }

    public static void main(String[] args) {
      try {
        NumarPozitiv n1 = new NumarPozitiv(3);
        NumarPozitiv n2 = new NumarPozitiv(5);
        System.out.println(n2.scadere(n1));
        System.out.println(n1.scadere(n2));
      } catch(ExceptieUnu e) {
        System.err.println(e.getMessage());
      } catch(ExceptieDoi e) {
        System.err.println(e.getMessage());
      } catch(Error error) {
        System.err.println(error.getMessage());
      }
    }
}
```

Constructorul clasei `NumarPozitiv` conţine o instrucţiune care aruncă o excepţie de tip `ExceptieUnu`. Deoarece aceasta nu are ca superclasă pe `RuntimeException` sau `Error`, trebuie neapărat să fie cuprinsă într-un bloc `try` cu o clauză `catch` care să prindă `ExceptieUnu` sau o superclasă a acesteia (`Exception` sau `Throwable`).

În metoda `scadere` sunt două instrucţiuni `throw` care aruncă `ExceptieDoi`, respectiv `Error`. Întrucât `ExceptieDoi` este derivată din `RuntimeException`, nu mai trebuie utilizată clauza `throws` sau o instrucţiune `catch`.

În metoda `main()`, clauzele `catch` pentru `ExceptieDoi` şi `Error` pot lipsi, în schimb cea pentru `ExceptieUnu` este obligatorie. Prima scădere va obţine valoare `2`, ceea ce este în regulă, însă la cea de-a doua scădere se obţine `-2` şi acest lucru va conduce la apariţia excepţiei `ExceptieDoi`. Deci la execuţie vom obţine:

```
2
Exceptia Doi
```

Dacă ar fi lipsit clauza `catch` pentru `ExceptieDoi` am fi obţinut:

```
2
Exception in thread "main" ExceptieDoi: Exceptia Doi
   at NumarPozitiv.scadere(NumarPozitiv.java:28)
   at NumarPozitiv.main(NumarPozitiv.java:45)
```

Observă în rezultat lista metodelor apelate în momentul apariţiei excepţiei.

3.2.13. Distrugerea obiectelor şi eliberarea memoriei

Obiectele (instanţele unei clase) se creează cu operatorul `new` sau cu metoda `newInstance()` a clasei `Class`. Marea majoritate a programelor Java în care se instanţiază clase folosesc operatorul `new`. Exemplul 2.4.8 pune în evidenţă instanţierea unei clase cu metoda `newInstance()`. În momentul creării unei instanţe pentru o clasă se alocă memorie pentru obiectul nou creat. Acesta va avea o adresă în memorie. Dacă obiectului creat i se aplică ulterior din nou operatorul `new` sau metoda `newInstance()`, atunci se alocă o nouă zonă de memorie care va avea bineînţeles altă adresă. Spaţiul vechi de memorie se eliberează automat. Acest mecanism se numeşte *colectarea gunoaielor* (engl. *garbage collection*), numit aşa în ideea reutilizării spaţiului de memorie.

Exemplul 3.2.62. Programul de mai jos pune în evidenţă realocarea memoriei pentru un obiect folosind operatorul `new` şi metoda `newInstance()` din clasa `Class`.

```
class C1 {
   int x = 10;
   C1() { System.out.println("Constructor fara argumente.");
```

```
    }
    C1(int x) {
      this.x = x;
      System.out.println("Constructor cu un argument.");
    }
  }
  public class TestDealocare {
    public static void main(String args[]) {
      C1 obiectUnu = new C1(8);
      System.out.println("adresa lui obiectUnu = " +
        obiectUnu.hashCode());
      obiectUnu = new C1(6);
      System.out.println("adresa lui obiectUnu = " +
        obiectUnu.hashCode());
      Class oClasa = null;
      try {
        oClasa = Class.forName("C1");
      }
      catch (ClassNotFoundException e) {
       System.err.println("Exceptie: n-am gasit clasa C1. " + e);
      }
      try {
        obiectUnu = (C1) oClasa.newInstance();
      }
      catch (InstantiationException e) {
        System.err.println(
          "Exceptie: nu putem instantia clasa C1." + e);
      }
      catch (IllegalAccessException e) {
        System.err.println(
          "Exceptie: acces ilegal la clasa C1." + e);
      }
      System.out.println("adresa lui obiectUnu = " +
        obiectUnu.hashCode());
    }
  }
```

Observăm că primul obiect (`obiectUnu`) al clasei `C1` a avut o viaţă foarte scurtă. Mai precis, s-a realocat alt spaţiu de memorie, iar vechiul spaţiu s-a eliberat, eventual pentru o reutilizare ulterioară. Al doilea obiect (numit tot `obiectUnu`) are repartizată altă zonă de memorie. Din nou, se realocă spaţiu pentru al treilea obiect numit la fel, urmând ca vechiul spaţiu să fie eliberat de către colectorul de gunoaie (engl. *garbage collector*). Astfel, programul va afişa la execuţie:

```
Constructor cu un argument.
adresa lui obiectUnu = 6129586
```

```
Constructor cu un argument.
adresa lui obiectUnu = 7852340
Constructor fara argumente.
adresa lui obiectUnu = 4711763
```

Pentru a distruge un obiect, urmând ca spaţiul de memorie să fie eliberat de *garbage collector*, îl asignăm cu `null`; astfel se pierde referinţa la acest obiect.

Exemplul 3.2.63. Considerăm următorul cod Java (presupunem că am definit o clasă `C1` care are o metodă publică `afiseaza()`):

```
C1 obiect;
obiect.afiseaza();
```

Programul se va opri la execuţie cu un mesaj de eroare de tip `null pointer exception`. Aceasta deoarece am declarat un obiect, dar nu l-am creat folosind `new` sau `newInstance()`. Obiectul nu există, el are valoarea `null`. Dacă dorim să distrugem explicit un obiect (deşi un obiect nefolosit este automat distrus de sistemul *garbage collector*), putem să îi asignăm valoarea `null` acelui obiect. Acesta va fi distrus automat la următorul apel al colectorului de gunoaie.

Există însă cazuri când alocarea zonei de memorie nu s-a făcut cu `new` sau `newInstance()`, de exemplu, metode (specificate native) scrise în alte limbaje de programare (cum ar fi C) care conţin apeluri ale funcţiilor `malloc()`, `calloc()` etc. De asemenea, există cazuri în care se solicită eliberarea unor alte resurse decât cele de memorie, de exemplu, a unor conexiuni de fişiere sau a unor conexiuni de reţea. Colectorul de gunoaie poate doar să elibereze automat memoria alocată cu operatorul `new` sau cu metoda `newInstance()`. Pentru a rezolva şi aceste cazuri speciale, Java pune la dispoziţie metoda:

```
protected void finalize() throws Throwable
```

moştenită din clasa `Object`. Vom defini această metodă pentru clasa cu care lucrăm. Când *garbage collector*-ul este gata să elibereze spaţiul de memorie folosit de obiect, acesta va apela mai întâi `finalize()` şi abia după aceea se va elibera memoria. Problema care se pune este că nu se ştie când va fi „curăţat" obiectul, deci nici când se va face apelul metodei `finalize()`. Se ştie doar că aceasta va avea loc înainte ca zona de memorie respectivă să fie reutilizată. Metoda `finalize()` nu poate fi apelată direct de programator. În clasa predefinită `System` există metodele statice:

```
public static void gc();
public static void runFinalization();
```

Metoda `gc()` forţează execuţia *garbage collector*-ului şi se utilizează împreună cu `runFinalization()`, care implică execuţia metodelor `finalize()` a obiectelor ce nu mai sunt referite de nici o referinţă, dar care nu au fost încă finalizate.

Exemplul 3.2.64. Programul de mai jos pune în evidenţă crearea unui număr mare de obiecte ale unei clase şi apoi eliberarea memoriei asociate acestora:

```
class TestGc {
  static boolean pornitColectorul = false;
  static boolean gata = false;
  static int numarInstante = 0;
  static int finalizate = 0;
  final int NR_INST_MAX = 32;
  int i;

  TestGc() {
    i = ++ numarInstante;
    if (numarInstante == NR_INST_MAX)
      System.out.println(
        "A fost creat numarul maxim de instante.");
  }

  protected void finalize() {
    if (!pornitColectorul) {
      pornitColectorul = true;
      System.out.println("Incepem sa finalizam dupa " +
        numarInstante + " instante create.");
    }
    if (i == NR_INST_MAX) {
      System.out.println("Finalizam ultimul obiect si " +
        "oprim crearea obiectelor de tip TestGc.");
      gata = true;
    }
    finalizate++;
    if(finalizate >= numarInstante)
      System.out.println("Toate cele " + finalizate +
        " obiecte au fost finalizate.");
  }
}

public class GarbageCollection {
  public static void main(String[] args) {
    while(!TestGc.gata) {
      new TestGc();
    }
    System.out.println("S-au creat in total " +
      TestGc.numarInstante + " instante; total finalizate = " +
      TestGc.finalizate);
    // urmeaza dealocarea memoriei
    System.out.println("Apelam gc():");
    System.gc();
    System.out.println("Apelam runFinalization():");
```

```
        System.runFinalization();
        System.out.println("Am terminat.");
    }
}
```

Clasa `TestGc` suprascrie metoda `finalize()` din clasa `Object`. Se va crea un număr mare de obiecte până când variabila booleană `gata` va deveni `true`, lucru care se întâmplă doar când colectorul de gunoaie va apela automat metoda `finalize()`. După ce variabila `gata` devine `true`, se opreşte iteraţia de creare a noilor obiecte din clasa `C1` şi se va afişa numărul total de obiecte create, respectiv finalizate. Astfel, o execuţie posibilă a programului de mai sus este (în funcţie de „viteza de reacţie" a colectorului de gunoaie):

```
A fost creat numarul maxim de instante.
Incepem sa finalizam dupa 34316 instante create.
Finalizam ultimul obiect si oprim crearea obiectelor de tip
TestGc
S-au creat in total 96138 instante; total finalizate = 91168
Apelam gc():
Toate cele 96138 obiecte au fost finalizate.
Apelam runFinalization():
Am terminat.
```

3.2.14. Divizarea unei aplicaţii în fişiere

Dacă un program constă doar dintr-o singură clasă, codul-sursă Java este plasat într-un fişier cu acelaşi nume ca al clasei şi cu extensia `.java`. Orice instrucţiune de tip import trebuie plasată înaintea declaraţiei clasei. Compilatorul transformă codul-sursă Java în cod binar Java, plasat într-un fişier cu numele clasei, dar cu extensia `.class`. Dacă programul constă în mai mult de o clasă există două alternative:

1. plasarea tuturor claselor într-un singur fişier;
2. plasarea fiecărei clase într-un fişier separat (cu acelaşi nume cu numele clasei).

Dacă toate clasele sunt plasate într-un singur fişier, prima clasă din fişier trebuie declarată `public`, iar celelalte nu. Numele fişierului trebuie să corespundă numelui clasei publice din respectivul fişier. În cele ce urmează, vom oferi o soluţie de a împărţi o clasă (eventual prea mare) în două clase de dimensiuni mai reduse. Presupunem că iniţial avem:

```
class A {
    ...
    public static void main(String args[]) {
        ...
        apel g();
        ...
    }
```

```
   void f()  {
      ...
   }
   void g() {
      ...
   }
}
```

Dorim ca metoda `g()` să fie în altă clasă, însă ea are acces la datele `private` şi `public` ale clasei `A`. O posibilă împărţire echivalentă în două clase este:

```
class A {
      ...
   B obiectDinB;
   public static void main(String args[]) {
      ...
      obiectDinB() = new B(this);
      obiectDinB.g();
   }
   void f() {
      ...
   }
}
class B {
   A obiectDinA;
      ...
   public B(A obiectDinA) {
      this.obiectDinA = obiectDinA;
   }
   void g() {
      //accesul la membrii public din A se face direct
      obiectDinA.membruPublicDinA;
      //accesul la datele private din clasa A se face folosind
      //functii de acces public
      obiectDinA.metodaPublica();
   }
}
```

Exemplul 3.2.65. Programul de mai jos pune în evidenţă o împărţire a unei clase în două clase echivalente mai mici:

```
public class C1 {
   public static void main(String args[]) {
      A obA = new A(1);
      B obB = new B(2, obA);
      obA = new A(1, obB);
      try {
        obB.get().afiseaza();
        obA.get().afiseaza();
      }
      catch (NullPointerException e) {
        System.err.println("Eroare: " + e.getMessage());
      }
    }
  }

  class A {
    private int x;
    private B obiectDinB;
    A(int x) {
      this.x = x;
      this.obiectDinB = null;
    }
    A(int x, B obiectDinB) {
      this.x = x;
      this.obiectDinB = obiectDinB;
    }
    B get() throws NullPointerException {
      if (obiectDinB == null)
        throw new NullPointerException(
          "Obiectul din B este null\n");
      else
        return obiectDinB;
    }
    void afiseaza() {
      System.out.println("x = " + x);
    }
  }

  class B {
    private int y;
    private A obiectDinA;

    B(int y) {
      this.y = y;
      this.obiectDinA = null;
    }
    B(int y, A obiectDinA) {
      this.y = y;
      this.obiectDinA = obiectDinA;
    }
    A get() throws NullPointerException {
      if (obiectDinA == null)
        throw new NullPointerException(
          "Obiectul din A este null\n");
      else
        return obiectDinA;
    }
    void afiseaza() {
      System.out.println("y = " + y);
    }
  }
```

Clasele `A`, respectiv `B` au ca atribute `x` şi `obiectDinB`, respectiv `y` şi `obiectDinA`. Cei doi constructori ai acestor clase au drept scop iniţializarea acestor atribute şi crearea legăturii duble între cele două instanţe ale claselor `A` şi `B`. În plus, clasele `A` şi `B` au câte o metodă de acces care returnează referinţa către obiectul celeilalte clase şi o metodă de tipărire a atributelor `x`, respectiv `y`. Pentru testarea acestui exemplu, clasa `C1` creează trei obiecte, două din clasa `A` (dintre care unul cu durată de viaţă foarte scurtă) şi unul din clasa `B`. Astfel, programul va afişa la execuţie:

```
x = 1
y = 2
```

3.3. Interfeţe

O *interfaţă* grupează mai multe metode şi date membre publice. Sunt definite doar prototipurile metodelor, implementările urmând a fi scrise în clasa care va implementa respectiva interfaţă. Gruparea metodelor publice este utilă pentru uniformizarea modalităţii de lucru cu anumite clase. Există şi interfeţe care nu posedă metode sau date membre, acestea având rol de indicator (engl. *flag*).

O clasă poate implementa mai multe interfeţe, dar poate moşteni doar o singură clasă. Relaţia de moştenire este valabilă şi pentru interfeţe. Prin implementarea mai multor interfeţe se simulează moştenirea multiplă (întâlnită în alte limbaje de programare orientate obiect, cum ar fi limbajul C++).

O declaraţie de interfaţă introduce un tip referinţă ai cărui membri sunt constante şi metode abstracte. Acest tip nu posedă implementări (metodele nu sunt definite), dar anumite clase pot implementa aceste interfeţe furnizând definiţii pentru metodele abstracte. Sintaxa generală a declaraţiei unei interfeţe este:

```
[public | abstract] interface <Identificator> [extends
<ListaInterfete>] {
    [<DeclaratiiMembriInterfata>]
}
```

unde:

- `<Identificator>` reprezintă numele interfeţei;
- `<ListaInterfete>` are forma generală: `<Interfata1>, ..., <InterfataN>`, unde `<InterfataI>` reprezintă identificatori (nume) de interfeţe, pentru `I` de la `1` la `N`;
- `<DeclaratiiMembriInterfata>` are forma generală: `<DeclaratieMembru1> ... <DeclaratieMembruK>`, unde `<DeclaratieMembruI>`, pentru `I` de la `1` la `K`, poate fi: `<DeclaratieConstanta>` sau `<DeclaratieMetoda Abstracta>`

Numele unei interfeţe (`<Identificator>`) nu poate coincide cu numele altei clase sau interfeţe din acelaşi pachet, altfel apare eroare la compilare. Ca şi în cazul claselor, domeniul numelui unei interfeţe este întreg pachetul în care este declarat. Fiecare interfaţă este implicit abstractă, deci este inutil modificatorul `abstract` în declaraţia unei interfeţe.

Dacă declaraţia unei interfeţe conţine clauza `extends`, atunci interfaţa va moşteni toate metodele şi constantele interfeţelor enumerate în `<ListaInterfete>`, care se numesc *superinterfeţele directe*. Orice clasă care implementează interfaţa declarată va trebui să implementeze toate superinterfeţele accesibile clasei.

Superinterfeţele din clauza `extends` trebuie să fie accesibile, altfel apare eroare la compilare. Nu există o superinterfaţă generală (pentru toate interfeţele), cum este clasa `Object` pentru superclase.

Ca şi în cazul claselor, relaţia de *derivare a interfeţelor* (indirectă) este închiderea tranzitivă a relaţiei de derivare directă (definiţia este dată în secţiunea 3.2.4).

Membrii unei interfeţe sunt acei membri moşteniţi din superinterfeţele directe şi acei membri declaraţi în interfaţă. Interfaţa moşteneşte toţi membrii superinterfeţelor, cu excepţia atributelor ascunse şi a metodelor suprascrise. Domeniul numelui unui membru declarat într-un tip interfaţă este corpul declaraţiei interfeţei. Toţi membrii unei interfeţe sunt implicit `public`. Astfel, dacă membrii sunt declaraţi `public`, atunci aceştia sunt accesibili şi în afara pachetului.

3.3.1. Declaraţiile atributelor unei interfeţe

Atributele unei interfeţe sunt, de fapt, constante şi au sintaxa generală:

```
[ public | static | final ] <Tip> <DeclaratieVariabila1> [, ...,
<DeclaratieVariabilaN>];
```

unde:

- modificatorii `public`, `static` şi `final` sunt impliciţi şi specificarea lor este inutilă;
- `<DeclaratieVariabilaI>`, cu `I = 1, ..., N`, poate fi:

```
<Identificator> = <InitializatorVariabila> sau
<Identificator> [] [ []...[] ] = <InitializatorTablou>
```

Spre deosebire de clasă, declaraţia unui atribut într-o interfaţă nu poate conţine modificatorii `synchronized`, `transient` sau `volatile`.

Fiecare atribut din corpul unei interfeţe trebuie să aibă o expresie de iniţializare, care nu este obligatoriu să fie constantă. Iniţializatorul variabilei este evaluat şi asignarea are loc o singură dată, când se iniţializează interfaţa. O expresie de iniţializare nu poate conţine o referinţă prin nume simplu la acelaşi atribut sau la alt atribut a cărui declaraţie apare mai târziu în aceeaşi interfaţă. La fel, dacă expresia de iniţializare conţine cuvintele rezervate `this` sau `super`, atunci apare o eroare la compilare.

Este posibil ca o interfaţă să moştenească mai multe atribute cu acelaşi nume de la mai multe interfeţe; încercarea de a ne referi cu nume simplu la un astfel de atribut conduce la eroare de compilare, deoarece referinţa este ambiguă.

Exemplul 3.3.1. Următorul program Java scoate în evidenţă situaţia când un atribut apare cu acelaşi nume în două interfeţe distincte:

```
interface I1 {
   int j = 3;
   int [] i = {j + 1, j + 2};
}

interface I2 {
   int j = 5;
   int [] i = {j + 2, j + 4};
}

public class TestAtribute implements I1, I2 {
   public static void main(String[] args) {
      System.out.println(I1.j);
      System.out.println(I2.i[1]);
      System.out.println(I1.i[0]);
   }
}
```

Dacă s-ar fi încercat accesarea atributelor `i` şi `j` din interfeţele `I1` şi `I2` fără numele calificat al interfeţelor, s-ar fi obţinut eroare la compilare. Programul este corect şi va afişa:

```
3
9
4
```

Dacă două atribute cu acelaşi nume sunt moştenite de o interfaţă, atunci ne confruntăm cu *ambiguitatea atributului*. Această situaţie poate apărea, de exemplu, când două superinterfeţe directe declară acest atribut. Încercarea de accesare a acestui atribut conduce la eroare de compilare. Evident, este posibilă accesarea folosind nume calificat (numele interfeţei dorite).

Invers, o altă situaţie este întâlnită când un singur atribut este moştenit de mai multe din aceeaşi interfaţă (se mai numeşte *moştenire multiplă*). Această situaţie se poate obţine, de exemplu, când interfaţa `I0` este superinterfaţă pentru alte două noi interfeţe, `I1` şi `I2`, care la rândul lor sunt superinterfeţe pentru o altă interfaţă `I3`. Astfel, atributele din interfaţa `I0` sunt moştenite de mai multe ori, însă se păstrează doar o singură copie, deci accesarea acestora nu determină eroare la compilare.

Exemplul 3.3.2. Următorul program Java exemplifică ambiguitatea şi moştenirea multiplă a atributelor:

```
interface I0 {
  int x = 3;
}

interface I1 extends I0 {
  int [] y = {x + 1, x + 2};
}

interface I2 extends I0 {
  int [] y = {x + 3, x + 4};
}

interface I3 extends I1, I2 {
  /* se face referire la primul element
     din tabloul y din interfata I1 */
  int z = I1.y[0];
  /* se face referire la al doilea element
     din tabloul y din interfata I2 */
  int t = I2.y[1];
}

public class TestAtributeAmbigue implements I3 {
  public static void main(String[] args) {
    System.out.println(x);
    System.out.println(z);
    System.out.println(t);
  }
}
```

Atributul `y` este prezent atât în interfaţa `I1`, cât şi în interfaţa `I2`. Accesarea simplă a acestuia în interfaţa `I3` şi în clasa `TestAtributeAmbigue` va conduce la eroare de compilare. Pentru a evita eroarea de compilare, trebuie utilizat numele interfeţei din care face parte atributul (`I1` sau `I2`). Atributul `x` este moştenit de două ori în interfaţa `I3`, deoarece `x` este prezent ca atribut în interfaţa `I1` şi `I2`. Astfel, programul va afişa la execuţie:

```
3
4
7
```

3.3.2. Declaraţiile metodelor unei interfeţe

Metodele unei interfeţe sunt asemănătoare cu metodele unei clase, cu excepţia faptului că structura `<CorpMetodă>` este `;` şi că nu permit apariţia tuturor modificatorilor. Spre deosebire de metodele unei clase, declaraţia unei metode într-o interfaţă nu poate conţine modificatorii `static`, `final`, `synchronized` sau `native`. Sintaxa generală a declaraţiei metodei unei interfeţe este:

```
[public | abstract] <TipRezultat> <Identificator>
 ([<ListaParametriFormali>])
       throws <TipClasa1> [, ..., <TipClasaN>];
```

unde:

- modificatorii `public` şi `abstract` sunt impliciţi şi specificarea lor este redundantă;
- `<Identificator>` reprezintă numele metodei.

Declaraţia unei metode *suprascrie* toate metodele cu aceeaşi signatură din superinterfeţele sale care sunt accesibile codului interfeţei. În acest caz, acestea trebuie să întoarcă acelaşi tip şi, dacă au clauze `throws`, nu trebuie să intre în conflict. Astfel, o interfaţă moşteneşte toate metodele nesuprascrise ale superinterfeţelor sale. Metodele declarate în interfeţe sunt abstracte şi deci nu au implementare. Cum metodele suprascrise au aceeaşi signatură (şi compilatorul le forţează să returneze acelaşi tip), înseamnă că singurul loc unde pot să difere este în lista de excepţii care pot fi aruncate de metoda suprascrisă (eventual restrângând lista acestora).

Exemplul 3.3.3. Următorul program Java evidenţiază suprascrierea metodelor unei interfeţe relativ la lista de excepţii care pot fi aruncate:

```
class ExceptiaNoastraUnu extends Exception {
  ExceptiaNoastraUnu() { super(); }
}

class ExceptiaNoastraDoi extends Exception {
  ExceptiaNoastraDoi() { super(); }
}

interface I1 {
  int f(int x) throws ExceptiaNoastraUnu, ExceptiaNoastraDoi;
}

interface I2 extends I1 {
  int f(int x) throws ExceptiaNoastraDoi;
}
```

```
class C1 implements I1 {
  public int f(int x) throws ExceptiaNoastraUnu,
  ExceptiaNoastraDoi {
    if (x < 0) throw new ExceptiaNoastraUnu();
    else throw new ExceptiaNoastraDoi();
  }
}

class C2 implements I2 {
  public int f(int x) throws ExceptiaNoastraDoi {
    if (x < 0) throw new ExceptiaNoastraDoi();
    return x;
  }
}

public class TestSuprascriereMetode {
  public static void main(String[] args) {
    C1 obiectUnu = new C1();
    C2 obiectDoi = new C2();
    try {
      System.out.println(obiectDoi.f(2));
      System.out.println(obiectUnu.f(-2));
      System.out.println(obiectDoi.f(-2));
      System.out.println(obiectUnu.f(2));
    }
    catch (ExceptiaNoastraUnu e) {
      System.err.println("A aparut o exceptie: " + e);
    }
    catch (ExceptiaNoastraDoi e) {
      System.err.println("A aparut o exceptie: " + e);
    }
  }
}
```

Metoda `f()` din clasa `C2` implementează metoda abstractă `f()` din interfaţa `I2`. Încercarea de a implementa în clasa `C2` metoda `f()` din interfaţa `I1` conduce la o eroare de compilare. Deşi clasa `TestSuprascriereMetode` presupune patru afişări de texte, se aruncă o excepţie încă de la a doua afişare la consola de ieşire. Astfel, programul va afişa la execuţie:

```
2
A aparut o exceptie: ExceptiaNoastraUnu
```

Dacă două metode ale unei interfeţe (declarate în aceeaşi interfaţă sau moştenite) au acelaşi nume, dar signaturi diferite, atunci spunem că numele metodei este *supraîncărcat*.

Exemplul 3.3.4. Următorul program Java scoate în evidenţă supraîncărcarea metodelor unei interfeţe:

```
interface I1 {
  int f(int x);
}

interface I2 extends I1 {
  float f(float x);
  double f(double x);
}

class C1 implements I2 {
  public int f(int x) { return x; }
  public float f(float x) { return x; }
  public double f(double x) { return x; }
}

public class TestSupraincarcareMetode {
  public static void main(String[] args) {
    C1 obiect = new C1();
    System.out.println(obiect.f(2));
    System.out.println(obiect.f(4.5));
    System.out.println(obiect.f(2.5f));
  }
}
```

Numele metodei `f()` este supraîncărcat cu trei signaturi diferite în interfaţa `I2`, clasa `C1` trebuind să implementeze toate cele trei metode. Astfel, programul va afişa:

```
2
4.5
2.5
```

3.3.3. Moştenire multiplă prin intermediul interfeţelor

Am văzut că Java nu permite moştenirea multiplă a claselor. Structura claselor este un arbore (şi nu un graf) cu rădăcină, în care o clasă poate avea oricâte subclase, dar numai o superclasă. Există, totuşi, un mod de simulare a facilităţilor moştenirii multiple în Java utilizând interfeţe. O clasă poate implementa orice număr de clase (interfeţe).

Exemplul 3.3.5. Programul Java de mai jos prezintă o aplicaţie în care se folosesc interfeţe. Avem o interfaţă `DispozitiveElectronice`, o clasă de bază `Locuinta`, două clase derivate (`TV` şi `Aspirator`) din clasa de bază. Fişierul de aplicaţii este intitulat `AplicatieElectronice.java`.

```
// Fisierul DispozitiveElectronice.java
interface DispozitiveElectronice {
  public void setTimpFolosinta(int timp);
  public int getTimpFolosinta();
  public int getPutere();
}

// Fisierul Locuinta.java
abstract class Locuinta {
  protected int volum = 1;
  private boolean on;

  public void porneste() {
    on = true;
  }

  public void opreste() {
    on = false;
  }

  public abstract void schimbaVolum(int diferenta);
}

// Fisierul TV.java
class TV extends Locuinta implements DispozitiveElectronice {
  private int timp = 0;

  public void schimbaVolum(int diferenta) {
    volum = volum + diferenta;
  }

  public void setTimpFolosinta(int timp) {
    this.timp = timp;
  }

  public int getTimpFolosinta() {
    return timp;
  }

  public int getPutere() {
    return 100;
  }

  public int getVolum() {
    return volum;
    }
}

// Fisierul Aspirator.java
class Aspirator extends Locuinta implements
    DispozitiveElectronice {
  private int timp = 0;

  public void schimbaVolum(int diferenta) {
    volum = volum + diferenta;
  }

  public void setTimpFolosinta(int timp) {
    this.timp = timp;
  }

  public int getTimpFolosinta() {
    return timp;
  }

  public int getPutere() {
    return 500;
  }

  public int getVolum() {
    return volum;
  }
}

// Fisierul AplicatieElectronice.java
import java.io.*;

public class AplicatieElectronice {
  private BufferedReader tastatura;
  private int totalPutere;
  private int timp, volum;

  public static void main(String[] args) {
    AplicatieElectronice obiect = new AplicatieElectronice();
    TV tv = new TV();
    obiect.citesteTimp();
    tv.setTimpFolosinta(obiect.timp);
    tv.schimbaVolum(1);
    obiect.calculeazaPutereConsumata(tv);
    obiect.afiseaza("tv", tv.getVolum(),
      tv.getTimpFolosinta(), obiect.totalPutere);
    tv.setTimpFolosinta(0);
    Aspirator aspirator = new Aspirator();
    obiect.citesteTimp();
    aspirator.setTimpFolosinta(obiect.timp);
    aspirator.schimbaVolum(2);
    obiect.calculeazaPutereConsumata(aspirator);
    obiect.afiseaza("aspirator", aspirator.getVolum(),
      aspirator.getTimpFolosinta(), obiect.totalPutere);
    aspirator.setTimpFolosinta(0);
  }

  private void citesteTimp() {
    try {
      tastatura = new BufferedReader(
        new InputStreamReader(System.in),1);
      System.out.print("Dati timpul (in minute): ");
      System.out.flush();
      String linie = tastatura.readLine();
      try {
        timp = Integer.parseInt(linie);
      }
      catch (NumberFormatException ex) {
       System.err.println("Format de intreg gresit: " + ex);
       System.exit(3);
      }
    }
    catch (IOException e) {
      System.err.println("Intrare gresita de la tastatura: "
         + e);
      System.exit(2);
    }
  }

  public void afiseaza(String s, int v, int timpUtilizat, int
  putere) {
    if (s.equals("tv")) {
      System.out.println("TV-ul are volumul " + v);
      System.out.println("si a fost utilizat timp de " +
      timpUtilizat + " minute.");
      System.out.println("Puterea totala a fost " + putere);
    }
    else
      if (s.equals("aspirator")) {
        System.out.println("Aspiratorul are viteza " + v);
```

```
         System.out.println("si a fost utilizat timp de " +
            timpUtilizat + " minute.");
        System.out.println("Puterea totala a fost " + putere);
       }
  }

  public void calculeazaPutereConsumata(DispozitiveElectronice
  dispozitiv) {
    int putere = dispozitiv.getTimpFolosinta() *
      dispozitiv.getPutere();
    totalPutere = totalPutere + putere;
  }
}
```

Interfaţa `DispozitiveElectronice` declară metode referitoare la timpul utilizat de un dispozitiv electronic (fiind metode de acces la atributele claselor implementate, s-a păstrat convenţia de notare cu set... şi get...) şi la puterea curentului aparatului electronic pe unitatea de măsură. Clasa abstractă `Locuinta` (este abstractă, deoarece nu implementează metoda `schimbaVolum()`) se referă la situaţia pornirii şi opririi unui aparat electronic. Clasele neabstracte `TV` şi `Aspirator` extind clasa (de bază) `Locuinta` şi implementează interfaţa `DispozitiveElectronice` prin definirea metodelor abstracte din acestea. Acestea au în comun atributul `volum` referitor la volumul/viteza utilizat(ă) de dispozitivul electronic (televizor/aspirator). În plus, clasele `TV` şi `Aspirator` au în particular un atribut `timp`. Chiar dacă un atribut provine de la superclasa `Locuinta` şi unul este propriu fiecărei clase, instanţele celor două clase nu au legătură comună referitoare la acestea (adică modificarea valorii acestui atribut pentru obiectul clasei `TV` nu afectează valoarea aceluiaşi atribut pentru clasa `Aspirator` şi reciproc).

Deoarece s-a dorit utilizarea doar a obiectelor, şi nu a metodelor statice, s-a creat un obiect al clasei `AplicatieElectronice`. Această clasă (de testare a aplicaţiei noastre) are ca atribute `timp`, `volum` şi, în plus faţă de clasele `TV` şi `Aspirator`, un atribut `totalPutere` pentru a păstra valoarea puterii consumate de dispozitivul electronic curent pe perioada de timp în care a fost utilizată. Se vor citi de la tastatură timpii de întrebuinţare a televizorului, respectiv aspiratorului, afişându-se volumul, timpul şi puterea totală consumată. O posibilă execuţie a programului de mai sus este:

```
Dati timpul (in minute): 60
TV-ul are volumul 2
si a fost utilizat timp de 60 minute.
Puterea totala a fost 6000
Dati timpul (in minute): 10
Aspiratorul are viteza 3
si a fost utilizat timp de 10 minute.
Puterea totala a fost 11000
```

3.4. Tablouri

În Java, *tablourile* sunt obiecte create dinamic și pot fi asignate variabilelor de tip `Object` (toate metodele clasei `Object` pot fi apelate într-un tablou). Un obiect tablou conține un număr de *variabile* (se mai numesc *elemente* sau *componente*), eventual zero elemente, caz în care se spune că tabloul este *vid*. Elementele unui tablou nu au nume, dar pot fi referențiate prin expresii de acces la tablou care se evaluează la valori întregi pozitive. Dacă un tablou are `n` elemente, spunem că `n` este lungimea tabloului (dat prin atributul `length`) și elementele pot fi accesate folosind indici de la `0` la `n-1`. Toate elementele unui tablou au același tip, numit *tipul componentelor* tabloului. Dacă tipul componentelor unui tablou este `<Tip>`, atunci tipul tabloului este `<Tip> []`.

Sintaxa generală a declarării unui tablou (fără crearea sau inițializarea acestuia) este:

```
<TipComponente> [[] ... []] <Identificator> [[] ... []];
```

unde:

- `<Identificator>` reprezintă numele tabloului;
- există cel puțin o pereche de paranteze pătrate (`[]`) și nu are importanță unde apar – înainte sau după `<Identificator>` (de regulă, se scriu înaintea identificatorului);
- `<TipComponente>` poate fi tip primitiv sau referință (sunt permise și tipurile interfață și clasă abstractă);
- tipul tabloului este `<TipComponente>` `[[]` ... `[]]`, unde numărul de `[]` este egal cu suma perechilor de paranteze pătrate care apar în declarație (numărul perechilor parantezelor se numește *dimensiunea* sau *adâncimea* tabloului).

Exemplul 2.7.4 evidențiază faptul că nu are importanță ordinea parantezelor. O variabilă de tip tablou memorează o referință de tip tablou. Declarația unei variabile de tip tablou nu creează întreg obiectul tablou sau alocă spațiu de memorie pentru componente, ci doar creează variabila care poate conține o referință la un tablou. Deoarece atributul `length` nu face parte din tipul tablou, o variabilă de tip tablou poate memora referințe la tablouri de diferite lungimi.

Un tablou se poate crea prin expresii de creare care folosesc operatorul `new` sau un inițializator de tablouri. O expresie de creare a unui tablou specifică tipul elementelor, dimensiunea tabloului (numărul de perechi de paranteze pătrate) și lungimea tabloului cel puțin pentru prima pereche de paranteze pătrate. Numărul de elemente ale tabloului este dat de variabila instanță constantă numită `length`.

Sintaxa generală de creare a unui tablou folosind operatorul `new` este:

```
<Identificator> = new <Tip> [<Expresie1>] [[<Expresie2>] ...
[<ExpresieN>]][[] ... []];
```

unde:

- `<Identificator>` este numele tabloului;
- `<Tip>` poate fi un tip primitiv sau referință;
- `<Expresie1>`, `<Expresie2>`, ..., `<ExpresieN>` sunt de tipuri convertibile la compilare la tipul `int`, altfel apare o eroare la compilare.

O expresie de creare a unui tablou este evaluată la execuție astfel (în ordine):

1. sunt evaluate `<Expresie1>`, `<Expresie2>`, ..., `<ExpresieN>`. Dacă vreo evaluare se termină anormal, atunci expresiile următoare nu mai sunt evaluate și programul se termină anormal;
2. sunt verificate valorile expresiilor `<Expresie1>`, `<Expresie2>`, ..., `<ExpresieN>` și dacă vreuna este găsită negativă, atunci se aruncă o excepție de tip `NegativeArraySizeException`;
3. se alocă pentru tablou spațiu de memorie, iar dacă nu există spațiu suficient pentru alocarea respectivă, atunci expresia de creare a tabloului se termină anormal, aruncând excepția `OutOfMemoryError`;
4. dacă `N = 1`, atunci se creează un tablou unidimensional de lungime specificată și fiecare componentă a tabloului este inițializată cu valoarea sa implicită (secțiunea 2.6);
5. dacă `N > 1`, atunci se creează un tablou de dimensiune `N` (un tablou de tablouri).

Exemplul 3.4.1. Instrucțiunea de creare a tabloului:

```
int [][] a = new int[3][4];
```

este echivalentă cu:

```
int [][] a = new int[3][];
for (int i = 0; i < a.length; i ++)
  a[i] = new int[4];
```

În acest caz, tabloul `a` va fi inițializat cu valoarea implicită a tipului `int`, adică 0.

Tablourile pot fi și neregulate, adică pot avea lungimi diferite pentru anumite dimensiuni. De exemplu, putem avea o matrice triunghiulară (adică alocăm memorie doar pentru elementele de sub diagonala principală).

Exemplul 3.4.2. Codul Java de mai jos:

```
int [][] a = new int[3][];
for (int i = 0; i < a.length; i ++)
  a[i] = new int[i + 1];
```

va aloca memorie pentru tabloul triunghiular `a`. Astfel, vor exista elementele `a[0][0]`, `a[1][0]`, `a[1][1]`, `a[2][0]`, `a[2][1]`, `a[2][2]`.

Într-o expresie de creare a unui tablou, expresiile dintre paranteze se evaluează de la stânga la dreapta.

Exemplul 3.4.3. Codul Java de mai jos:

```
public class Test {
    public static void main(String[] args) {
      int i = 3;
      int [][] a = new int[i][i = 2];
      System.out.println("numar linii = " + a.length +
        ", numar coloane = " + a[0].length);
    }
}
```

va aloca memorie pentru tabloul a de dimensiuni `[3][2]`, deoarece se evaluează întâi expresia `i`, apoi `i = 2`. Astfel, se va afişa:

```
numar linii = 3, numar coloane = 2
```

Un iniţializator de tablouri furnizează valori iniţiale pentru toate componentele sale. Sintaxa generală de creare a unui tablou folosind un iniţializator de tablou este:

```
<Identificator> = <InitializatorTablou>;
```

iar `<InitializatorTablou>` are forma generală:

```
{[<InitializatorVariabila1>] [, <InitializatorVariabila2>]...
[, <InitializatorVariabilaN>] }
```

unde:

- `<Identificator>` este numele tabloului;
- `<InitializatorVariabila1>`, `<InitializatorVariabila2>`, ..., `<InitializatorVariabilaN>` pot fi `<Expresie>` sau `<InitializatorTablou>`.

Lungimea tabloului construit va fi egală cu numărul de expresii. Fiecare expresie specifică o valoare pentru o componentă a tabloului şi trebuie să fie compatibilă cu tipul componentei tabloului (în caz contrar, apare o eroare de compilare).

Exemplul 3.4.4. Codul Java de mai jos pune în evidenţă situaţia când se încearcă accesarea unei referinţe `null`:

```
public class TestInitializareTablou {
  public static void main(String[] args) {
    int a[][] = { {3, 4}, null };
      for (int i = 0; i < 2; i++)
        for (int j = 0; j < 2; j++)
          System.out.println(a[i][j]);
  }
}
```

Acesta va afişa la execuţie:

```
3
4
java.lang.NullPointerException
    at TestInitializareTablou.main(TestInitializareTablou.java:6)
```

O componentă a unui tablou este accesată de o expresie de acces a unui tablou a cărui valoare este o referinţă la un tablou urmat de o expresie index încadrată de paranteze pătrate. Un tablou de lungime `n` poate fi indexat de întregii `0`, `1`, ..., `n-1`. Tablourile pot fi indexate de valori de tip `int` sau care pot fi convertite la `int` (de exemplu, valorile `short`, `byte` şi `char`). În schimb, încercarea de accesare a unei componente tablou cu un index de tip `long` va genera o eroare la compilare. Accesul la tablouri este verificat în momentul execuţiei; orice încercare de utilizare a unui index mai mic decât `0` sau mai mare decât lungimea tabloului (dat prin atributul `length`) va implica aruncarea excepţiei `IndexOutOfBoundsException`.

Exemplul 3.4.5. Codul Java de mai jos pune în evidenţă situaţia când se încearcă accesarea unei componente cu un index prea mare:

```
class TestAccesTablou {
  public static void main(String[] args) {
    int[] v = new int[3];
    for (int i = 0; i < v.length; i++)
      v[i] = i;
    System.out.println(v[v.length]);
  }
}
```

Deoarece tabloul `v` este indexat de la `0` la `v.length-1`, programul de mai sus va afişa la execuţie:

```
java.lang.ArrayIndexOutOfBoundsException
      at TestAccesTablou.main(TestAccesTablou.java:6)
```

Membrii unui tip tablou sunt:

1. atributul `public final int length`, care conţine numărul de componente ale tabloului;
2. metoda `public Object clone()`, care suprascrie metoda cu acelaşi nume din clasa `Object`;
3. toţi membrii moşteniţi din clasa `Object` (cu excepţia lui `clone()` care este suprascrisă).

Exemplul 3.4.6. Codul Java de mai jos ilustrează crearea unei alte variabile de tip tablou care reprezintă o copie:

```
class TestClona {
  public static void main(String[] args) {
    int v1[] = {3, 4};
    int v2[] = (int[]) v1.clone();
    System.out.println(v1.hashCode() + " " + v2.hashCode());
    System.out.print((v1 == v2) + " ");
    v1[1]++;
```

```
        System.out.println(v2[1]);
    }
}
```

Variabilele de tip tablou `v1` şi `v2` nu au aceeaşi adresă, deci modificarea ulterioară a originalului nu va implica şi modificarea copiei. Pentru justificarea celor spuse vom apela metoda `hashCode()`, obţinând astfel la execuţia programului de mai sus:

```
7474923 3242435
false 4
```

O clonă a unui tablou multidimensional va conţine doar adresa de bază distinctă şi creează doar un nou tablou, subtablourile fiind comune cu tabloul iniţial.

Exemplul 3.4.7. Codul Java de mai jos evidenţiază crearea unei alte variabile de tip tablou bidimensional care reprezintă o copie:

```
class TestClonaTablouriBidimensionale {
  public static void main(String[] args) throws Throwable {
    int a[][] = {{1, 2}, {3, 4}};
    int copie[][] = (int[][])a.clone();
    a[0][1] = 5;
    System.out.print(copie[0][1] + " ");
    System.out.print((a == copie) + " ");
    System.out.println(a[0] == copie[0] && a[1] == copie[1]);
  }
}
```

Adresa de bază va fi distinctă, în schimb tablourile `a[0]`, `a[1]` sunt aceleaşi cu `copie[0]`, respectiv `copie[1]`. Modificarea originalului va implica şi modificarea copiei şi reciproc. Programul de mai sus va afişa la execuţie:

```
5 false true
```

Fiecare tablou are un obiect `Class` asociat, care este comun tuturor tablourilor având componente de acelaşi tip şi acelaşi număr de dimensiuni. Instanţele clasei `Class` reprezintă clasele şi interfeţele definite într-o aplicaţie Java.

Superclasa tipului tablou este clasa `Object`.

Exemplul 3.4.8. Codul Java de mai jos pune în evidenţă afişarea clasei şi superclasei unui tablou unidimensional:

```
class TestSuperclasa {
  public static void main(String[] args) {
    int[] v = new int[10];
    System.out.println(v.getClass());
    System.out.println(v.getClass().getSuperclass());
  }
}
```

Ca şi în exemplul 2.6.1, la afişarea clasei curente se va folosi o paranteză pătrată urmată de litera `I`, care semnifică tipul `int`, superclasa tablourilor fiind clasa `Object`. Astfel, programul va afişa la execuţie:

```
class [I
class java.lang.Object
```

Spre deosebire de alte limbaje de programare, un tablou de caractere nu este totuna cu un obiect `String`. Conţinutul unui obiect `String` nu se poate schimba pe componente (engl. *immutable*), pe când elementele unui tablou se pot schimba pe componente. Vom vedea în capitolele ce urmează că există în clasa `String` metoda `toCharArray()` care returnează tabloul de caractere ce conţine aceeaşi secvenţă de caractere ca şi obiectul `String`. Clasa `StringBuffer` conţine metode necesare pentru modificarea componentelor (engl. *mutable*).

Continuăm cu prezentarea transmiterii tablourilor ca parametri. Deoarece tablourile sunt de fapt nişte obiecte, transmiterea lor ca parametri se realizează via apel prin referinţă. Deci când un tablou se transmite ca parametru unei metode, se trimite o copie a adresei tabloului (se mai cheamă *referinţă a tabloului*). O referinţă nu este o copie a tabloului, ci un pointer către tablou. Deci când o metodă schimbă parametrul formal de tip tablou, acesta schimbă şi parametrul actual din apelul metodei.

Exemplul 3.4.9. Codul Java de mai jos pune în evidenţă transmiterea unui tablou unidimensional ca parametru pentru o metodă:

```
public class TransmitereTablouriParametri {

  public static void main(String args[]) {
    int [] a = new int[5];
    afiseazaSir(a);
    initializeazaSir(a);
    afiseazaSir(a);
  }

  static void initializeazaSir(int [] sir) {
    for(int i = 0; i < sir.length; i++)
      sir[i] = 1;
  }

  static void afiseazaSir(int [] sir) {
    for(int i = 0; i < sir.length; i++)
      System.out.print(sir[i] + " ");
    System.out.println();
  }
}
```

La început, tabloul `a` se iniţializează de către sistem cu valori implicite (adică `0`). Apelul metodei statice `afiseazaSir()` va transmite o copie a adresei tabloului `a`.

Metoda `afiseazaSir()` nu va modifica valorile elementelor parametrului formal, adică ale tabloului `sir`, și deci nici pe cele ale tabloului `a`. În schimb, metoda `initializeazaSir()` va atribui literalul `1` ca valoare pentru componentele tabloului `sir`, deci și pentru cele ale tabloului `a`. Astfel, programul Java de mai sus va afișa la execuție:

```
0 0 0 0 0
1 1 1 1 1
```

Exemplul 3.4.10. Codul Java de mai jos evidențiază faptul că la transmiterea unui tablou ca parametru se trimite o copie a adresei sale (și nu originalul):

```
public class TestAdresaTablouri {

  public static void main(String args[]) {
    int v[] = {2, 3};
    System.out.println("Inainte de apel, adresa lui v este " +
      v.hashCode());
    afiseazaTablou(v);
    f(v);
    System.out.println("Dupa apel, adresa lui v este " +
      v.hashCode());
    afiseazaTablou(v);
  }

  public static void f(int [] sir) {
    int w[] = {5, 6};
    sir = w;
    System.out.println("In metoda f(), adresa lui sir este " +
      sir.hashCode());
    afiseazaTablou(sir);
  }

  static void afiseazaTablou(int [] sir) {
    System.out.print("Tabloul este: ");
    for(int i = 0; i < sir.length; i++)
      System.out.print(sir[i] + " ");
    System.out.println();
  }
}
```

În metoda `main()`, tabloul `v[]` are o adresă în memorie (pe care o notăm cu `A`). În momentul apelului `f(v)` se va transmite o copie a adresei lui `A` ca parametru pentru tabloul `sir`. Tabloul `w` va avea o altă adresă în memorie (pe care o notăm cu `B`). La asignarea `sir = w`, variabila `sir` va avea adresa `B`, astfel că modificările asupra lui `w` se vor vedea şi în `sir`. Însă la terminarea execuţiei metodei `f()`, variabila `w` (deci şi variabila `sir`) va fi inaccesibilă. Adresa tabloului `v` din memorie nu s-a schimbat, deci este tot `A`. Astfel, programul va afişa la execuţie:

```
Inainte de apel, adresa lui v este 7474923
Tabloul este: 2 3
In metoda f(), adresa lui sir este 3242435
Tabloul este: 5 6
Dupa apel, adresa lui v este 7474923
Tabloul este: 2 3
```

Exemplul 3.4.11. Codul Java de mai jos ilustrează transmiterea (şi iniţializarea) tablourilor bidimensionale ca parametri:

```
public class MatriceParametri {

  public static void main(String args[]) {
    int [][] v1 = new int[2][3];
    initializeazaUnu(v1);
    afiseazaMatrice(v1);
    int [][] v2 = { {1, 3, 4}, {4, 6, 3} };
    afiseazaMatrice(v2);
  }

  static void afiseazaMatrice(int [][] a) {
    for (int i = 0; i < a.length; i++) {
      for (int j = 0; j < a[0].length; j++)
        System.out.print(a[i][j] + " ");
      System.out.println();
    }
  }

  static void initializeazaUnu(int [][] a) {
    int numarLinii = a.length;
    int numarColoane = a[0].length;
    for (int linie = 0; linie < numarLinii; linie++)
     for (int coloana = 0; coloana < numarColoane; coloana++)
        a[linie][coloana] = 1;
  }
}
```

Programul va afişa la execuţie:

```
1 1 1
1 1 1
1 3 4
4 6 3
```

3.5. Conversii ale tipului referinţă

În această secţiune ne vom referi la conversii ale tipurilor clasă, interfaţă şi tablou. În secţiunea 2.8 am prezentat conversiile identice, primitive (implicite şi explicite) şi pe cea implicită la `String`. Au rămas de prezentat conversiile de referinţă (implicite şi explicite), cele explicite la `String`, cele implicite de la clasele *wrapper* la tipurile primitive corespondente şi invers, de la tipurile primitive la clasele *wrapper*.

3.5.1. Conversii implicite ale tipului referinţă

Asemenea conversii nu necesită acţiuni speciale în momentul execuţiei, deci nu aruncă excepţii la execuţie. Se semnalează încă de la compilarea programului Java dacă o referinţă se poate converti implicit la alt tip. Notând cu `A` tipul referinţă sursă (clasă, interfaţă sau tablou) şi cu `B` tipul referinţă destinaţie, la care se verifică în faza de compilare conversia implicită, distingem următoarele situaţii posibile:

1. Dacă `A` este un tip clasă, atunci:
   a) Dacă `B` este un tip clasă, atunci `A` trebuie să coincidă cu `B` sau `A` trebuie să fie subclasă a lui `B`, altfel apărând o eroare de compilare;
   b) Dacă `B` este un tip interfaţă, atunci `A` trebuie să implementeze interfaţa `B`, altfel apare o eroare de compilare;
   c) Dacă `B` este un tip tablou, atunci apare o eroare de compilare.
2. Dacă `A` este un tip interfaţă, atunci:
   a) Dacă `B` este un tip clasă, atunci `B` trebuie să fie clasa `Object`, altfel apare o eroare de compilare;
   b) Dacă `B` este un tip interfaţă, atunci `A` trebuie să coincidă cu `B` sau `A` trebuie să fie subinterfaţă a lui `B`, altfel apare o eroare de compilare;
   c) Dacă `B` este un tip tablou, atunci apare o eroare de compilare.
3. Dacă `A` este un tip tablou (de forma `AA[]`), atunci:
   a) Dacă `B` este un tip clasă, atunci `B` trebuie să fie clasa `Object`, altfel apare o eroare de compilare;
   b) Dacă `B` este un tip interfaţă, atunci `B` trebuie să coincidă cu `Cloneable` (singura interfaţă implementată de tablouri), altfel apare o eroare de compilare;
   c) Dacă `B` este un tip tablou (de forma `BB[]`), atunci întâlnim una dintre situaţiile:
      i. `AA` şi `BB` reprezintă acelaşi tip primitiv;
      ii. `AA` şi `BB` sunt tipuri referinţă şi `AA` este asignabil lui `BB`;
      iii. Dacă primele două nu pot avea loc, atunci apare o eroare de compilare.

Ideea principală a conversiilor implicite este că acestea se realizează de la particular la general. De exemplu, dacă `C2` este o clasă Java şi `C1` superclasa sa, se poate face conversia implicită de la un obiect din `C2` la un obiect din `C1`, deoarece clasa `C1` are mai puţini membri decât `C2` (care moşteneşte toţi membrii din `C1`).

Exemplul 3.5.1. Următorul program Java pune în evidenţă câteva din situaţiile posibile prezentate mai sus:

```
class C1 { int x; }

class C2 extends C1 { int y; }

interface I1 { void f(int x); }

class C3 extends C1 implements I1 {
  int x;
  public void f(int x) { this.x = x; }
}

public class ConversiiReferintaCompilare {
  public static void main(String[] args) {
    // Asignari de variabile de tip clasa
    C1 obiectUnu = new C1();
    // asignarea este posibila deoarece C2 este subclasa a lui C1
    obiectUnu = new C2();
    // reciproc, asignarea nu este posibila deoarece este necesara
    // o conversie explicita de forma (C2)
    C2 obiectDoi = obiectUnu;
    // Asignari de variabile de tip Object
    // asignarea este posibila deoarece C1 este subclasa a lui
    // Object
    Object obiectTrei = obiectUnu;
    int[] a = new int[3];
    // asignarea este posibila deoarece clasa tablourilor este
    // subclasa a lui Object
    Object obiectPatru = a;
    // Asignari de variabile de tip interfata
    C3 obiectCinci = new C3();
    // asignarea este posibila deoarece clasa C3 implementeaza I1
    I1 obiectSase = obiectCinci;
    // Asignari de variabile de tip tablou
    byte[] b = new byte[4];
    // asignarea nu este posibila deoarece tablourile a si b
    // nu sunt definite pe acelasi tip primitiv
    a = b;
    C2 [] obiectSapte = new C2[3];
    // asignarea este posibila deoarece C2 este subclasa a lui C1
    C1 [] obiectOpt = obiectSapte;
    // asignarea nu este posibila fiind necesara o conversie
    // explicita la (C2 [])
```

```
        obiectSapte = obiectOpt;
    }
}
```

Execuţia acestui program va genera trei erori de compilare (menţionate şi în comentariile programului).

3.5.2. Conversii explicite ale tipului referinţă

Aceste conversii necesită un test suplimentar la momentul execuţiei pentru a afla dacă valoarea referinţei actuale este o valoare legitimă a acestui tip nou, în caz contrar se aruncă o excepţie `ClassCastException`. Notăm cu `A` tipul referinţă sursă (clasă, interfaţă sau tablou) şi cu `B` tipul referinţă destinaţie. Sintactic, conversia explicită necesită scrierea lui `B` între paranteze rotunde (adică `(B)`). O conversie explicită poate fi determinată ca fiind corectă încă de la faza de compilare. O conversie de la `A` la `B` este corectă la faza de compilare dacă şi numai dacă `A` poate fi convertit la `B` printr-o conversie de asignare (secţiunea 2.8.1).

Prezentăm mai întâi situaţiile de conversie explicită care se verifică în faza de compilare:

1. Dacă `A` este un tip clasă, atunci:
   a) Dacă `B` este un tip clasă, atunci `A` trebuie să coincidă cu `B` sau `A` trebuie să fie subclasă a lui `B`, sau `B` trebuie să fie subclasă a lui `A`, altfel apare o eroare de compilare;
   b) Dacă `B` este un tip interfaţă, atunci:
      i. Dacă `A` este o clasă nedeclarată `final`, atunci conversia explicită este corectă la compilare;
      ii. Dacă `A` este o clasă declarată `final`, atunci `A` trebuie să implementeze `B`, altfel apare o eroare la compilare;
   c) Dacă `B` este un tip tablou, atunci `A` trebuie să fie clasa `Object`, altfel apare o eroare de compilare.
2. Dacă `A` este un tip interfaţă, atunci:
   a) Dacă `B` este o clasă nedeclarată `final`, atunci conversia explicită este corectă la compilare;
   b) Dacă `B` este o clasă declarată `final`, atunci `B` trebuie să implementeze `A`, altfel apare o eroare la compilare;
   c) Dacă `B` este o interfaţă şi `A`, `B` conţin metode cu aceeaşi signatură, dar tipuri returnate diferite, atunci apare o eroare de compilare.
3. Dacă `A` este un tip tablou (de forma `AA[]`), atunci:
   a) Dacă `B` este un tip clasă, atunci `B` trebuie să fie clasa `Object`, altfel apare o eroare de compilare;
   b) Dacă `B` este un tip interfaţă, atunci `B` trebuie să coincidă cu `Cloneable` (singura interfaţă implementată de tablouri), altfel apare o eroare de compilare;
   c) Dacă `B` este un tip tablou (de forma `BB[]`), atunci are loc unul dintre cazurile:

i. `AA` şi `BB` reprezintă acelaşi tip primitiv;
ii. `AA` şi `BB` sunt tipuri referinţă şi `AA` este convertit (implicit) la `BB`;
iii. Dacă ii. şi iii. nu pot avea loc, atunci apare o eroare de compilare.

Exemplul 3.5.2. Următorul program Java pune în evidenţă câteva dintre situaţiile posibile discutate mai sus:

```
class C1 { int x; }

class C2 extends C1 { int y; }

interface I1 { void f(int x); }

class C3 extends C1 implements I1 {
  int x;
  public void f(int x) { this.x = x; }
}

public class ConversiiReferintaCompilareExplicite {
  public static void main(String[] args) {
    // Asignari de variabile de tip clasa
    C1 obiectUnu = new C1();
    C2 obiectDoi = new C2();
    // asignarea este posibila deoarece C2 este subclasa a lui C1
    obiectUnu = obiectDoi;
    // reciproc, asignarea este posibila deoarece este necesara
    // o conversie explicita de forma (C2)
    obiectDoi = (C2) obiectUnu;
    // Asignari de variabile de tip interfata
    C3 obiectTrei = new C3();
    // asignarea este posibila la compilare deoarece este
    // facuta o conversie explicita. Totusi, depinde de tipul
    // de la executie al lui obiectUnu. Este necesara o
    // verificare la executie a tipului lui obiectUnu.
    // obiectUnu poate sa nu apartina nici lui
    // C3, nici unei subclase a acestuia.
    obiectTrei = (C3) obiectUnu;
    // asignarea este posibila deoarece clasa C3 implementeaza I1
    I1 obiectPatru = obiectTrei;
    // La fel ca mai sus, asignarea este posibila
    // la compilare, insa nu mai este posibila
    // la executie, deoarece obiectUnu
    // poate sa nu implementeze interfata I1.
    obiectPatru = (I1) obiectUnu;
  }
}
```

Programul este corect la compilare, însă la execuție va arunca excepția `ClassCastException`:

```
Exception in thread "main" java.lang.ClassCastException: C2
     at ConversiiReferintaCompilareExplicite.main (Conversii
     ReferintaCompilare Explicite.java:26)
```

Exemplul 3.5.3. Următorul program Java pune în evidență situațiile de conversie referitoare la tablouri:

```
class C1 { int x; }

class C2 extends C1 { int y; }

public class ConversiiTablouriCompilareExplicite {
  public static void main(String[] args) {
    // Asignari de variabile de tip tablou
    int [] a = new int[5];
    // asignarea este posibila deoarece int[] este subclasa a
    // lui Object
    Object o = a;
    // Valoarea lui a nu se poate asigna lui b, deoarece a
    // este tablou cu componente de tip primitiv, pe cand
    // Integer nu este tip primitiv.
    // Se obtine eroare la compilare.
    Integer [] b = a;
    // Valoarea lui a nu se poate asigna lui c, deoarece a si
    // c sunt tablouri cu componente de tipuri diferite.
    // Se obtine eroare la compilare.
    short [] c = a;
    C1 [] obiectUnu = new C1[10];
    C2 [] obiectDoi = new C2[11];
    // asignarea este corecta deoarece C2 este subclasa a lui C1
    obiectUnu = obiectDoi;
    // asignarea este corecta la faza de compilare, dar va arunca
    // o exceptie ArrayStoreException la executie
    obiectUnu[0] = new C1();
    // asignarea nu este posibila, se obtine eroare la compilare
    obiectDoi = obiectUnu;
  }
}
```

Așa cum s-a menționat și în comentariile programului, sunt trei erori la compilare și o eroare la execuție.

Continuăm cu prezentarea situațiilor de conversie explicită care se verifică în faza de execuție, dar care nu au fost detectate (nu au putut fi verificate) la compilare.

Anumite conversii explicite necesită o verificare la momentul execuţiei. Notăm cu `A` tipul referinţă sursă, cu `B` tipul referinţă destinaţie şi cu `C` clasa obiectului referit de valoarea referinţei la execuţie. Dacă valoarea de la execuţie este `null`, atunci este permisă conversia. Altfel, conversia explicită trebuie să verifice la execuţie dacă tipul obiectului (clasa `C`) este compatibil cu `B` la asignare folosind algoritmul de mai sus (cel aplicat la faza de compilare). Iată situaţiile de conversie explicită care se verifică în faza de execuţie:

1. Dacă `C` este un tip clasă obişnuită (deci nu o clasă tablou), atunci:
   a) Dacă `B` este un tip clasă, atunci `C` trebuie să coincidă cu `B` sau `C` trebuie să fie subclasă a lui `B`, altfel se aruncă o excepţie de tip `ClassCastException` la execuţie;
   b) Dacă `B` este un tip interfaţă, atunci `C` trebuie să implementeze `B`, altfel se aruncă o excepţie de tip `ClassCastException` la execuţie;
   c) Dacă `B` este un tip tablou, atunci se aruncă o excepţie de tip `ClassCastException` la execuţie.
2. Dacă `C` este un tip interfaţă, atunci:
   a) Dacă `B` este o clasă, atunci `B` trebuie să fie clasa `Object`, altfel se aruncă o excepţie de tip `ClassCastException` la execuţie;
   b) Dacă `B` este o interfaţă, atunci `C` trebuie să fie aceeaşi interfaţă cu `B` sau o subinterfaţă a lui `B`, altfel se aruncă o excepţie de tip `ClassCastException` la execuţie;
   c) Dacă `B` este un tip tablou, atunci se aruncă o excepţie de tip `ClassCastException` la execuţie.
3. Dacă `C` este un tip tablou (de forma `CC[]`), atunci:
   a) Dacă `B` este o clasă, atunci `B` trebuie să fie clasa `Object`, altfel se aruncă o excepţie de tip `ClassCastException` la execuţie;
   b) Dacă `B` este un tip interfaţă, atunci `B` trebuie să coincidă cu `Cloneable` (singura interfaţă implementată de tablouri), altfel se aruncă o excepţie de tip `ClassCastException` la execuţie;
   c) Dacă `B` este un tip tablou (de forma `BB[]`), atunci una dintre afirmaţiile următoare este adevărată:
      i. `BB` şi `CC` reprezintă acelaşi tip primitiv;
      ii. `BB` şi `CC` sunt tipuri referinţă şi `CC` este convertibil (implicit) la `BB`;
      iii. Dacă ii. şi iii. nu pot avea loc, atunci se aruncă o excepţie de tip `ClassCastException` la execuţie.

Exemplul 3.5.4. Următorul program Java ilustrează unele situaţii de conversie care se verifică la execuţie (liniile sunt numerotate):

```
1. class C1 { int x; }
2. interface I1 { void f(int x); }
3. class C2 extends C1 implements I1 {
4.   int x;
5.   public void f(int x) { this.x = x; }
6. }
```

```
 7. public class TestConversie1 {
 8.   public static void main(String[] args) {
 9.     C1 obiectUnu = new C1();
10.     C2 obiectDoi = new C2();
11.     I1 obiectTrei;
12.     obiectDoi = (C2)obiectUnu;
13.     obiectUnu = obiectDoi;
14.     obiectTrei = (I1)obiectUnu;
15.     Long x = (Long)obiectUnu;
16.   }
17.}
```

Linia 12 va genera eroare la execuţie, deoarece variabila `obiectUnu` poate să nu aparţină lui `C2` sau unei clase derivate. Linia 14 este corectă şi la compilare şi la execuţie, deoarece clasa `C2` implementează interfaţa `I1` (dacă ar fi lipsit linia 13, am fi obţinut eroare la execuţie). Linia 15 va afişa o eroare la compilare, fiindcă tipurile `Long` şi `C2` (şi `C1`) nu au nimic în comun, deci nu se poate face nici conversie explicită.

Exemplul 3.5.5. Următorul cod Java evidenţiază situaţiile de conversie care se verifică la execuţie (liniile sunt numerotate):

```
 1. class C1 {
 2.   int x = 1;
 3.   C1 (int x) { this.x = x; }
 4. }
 5. class C2 extends C1 {
 6.   int x = 3, y = 4;
 7.   C2 (int x, int y) { super(x); this.y = y; }
 8. }
 9. public class TestConversie2 {
10.   public static void main(String args[]) {
11.     C1[] tablouUnu = new C2[5];
12.     for (int i = 0; i < tablouUnu.length; i++)
13.       tablouUnu[i] = new C2(i * 2, i * 3);
14.     C2[] tablouDoi = (C2[]) tablouUnu;
15.     for (int i = 0; i < tablouUnu.length; i++)
16.     System.out.print(tablouDoi[i].x + " " + tablouDoi[i].y
                + " ");
17.   }
18.}
```

Linia 11 defineşte variabila `tablouUnu` ca având la compilare tipul `C1[]`, iar la execuţie `C2[]`. Linia 14 specifică variabila `tablouDoi` ca având tipurile de compilare şi execuţie `C2[]` (datorită conversiei explicite). Dacă ar fi lipsit conversia explicită, este clar că s-ar fi obţinut eroare la compilare la linia 14 (conform regulilor de mai sus). Astfel, programul de mai sus este corect şi va afişa la execuţie:

```
3 0 3 3 3 6 3 9 3 12
```

Presupunem acum că linia 11 ar fi înlocuită cu:

```
11.     C1[] tablouUnu = new C1[5];
```

Atunci variabila `tablouUnu` are la compilare şi execuţie tipul `C1[]`. Programul este corect la compilare, însă la execuţie va arunca o excepţie de forma `ClassCastException` la linia 14, deoarece se aplică algoritmul de conversie discutat mai sus (punctul 3. c) ii).

Continuăm cu prezentarea accesului atributelor şi metodelor la clasa curentă şi la superclasele acesteia folosind conversii explicite. Astfel, fie `C1` clasa de bază şi `C2` o clasă derivată a lui `C1`. Ştim că se poate defini un obiect din clasa `C1` care să fie instanţă pentru `C2`, astfel:

```
C1 obiect = new C2();
```

Pentru rezolvarea ambiguităţii (eventuale a) membrilor suprascrişi ai lui `obiect`, în mod implicit, atributele lui `obiect` sunt cele din `C1`, iar metodele lui `obiect` sunt cele din `C2`. Accesarea atributelor lui `obiect` din clasa `C2` se face prin conversie explicită. Exemplele 3.2.26, 3.2.27 şi 3.2.36 pun în evidenţă şi utilizarea altor procedee (prin nume calificat, întrebuinţând cuvântul rezervat `super`).

Exemplul 3.5.6. Programul Java următor ilustrează situaţiile de mai sus:

```
class C1 {
  int x = 1;
  void f(int x) {
    System.out.print("1 ");
    this.x = x;
  }
}

class C2 extends C1 {
  int x = 2;
  public void f(int x) {
    System.out.print("2 ");
    this.x = x;
  }
}

public class TestMic {
  public static void main(String[] args) {
    C1 obiectUnu = new C2();
    System.out.print(obiectUnu.x + " ");
    obiectUnu.f(4);
   System.out.print(obiectUnu.x + " " + ((C2)obiectUnu).x + " ");
  }
}
```

Variabila `obiectUnu.x` se va referi la atributul `x` din clasa `C1`, iar `obiectUnu.f()` la metoda `f()` din clasa `C2`. Obţinerea atributului `x` din clasa `C2` se poate face prin conversie explicită astfel: `((C2)obiectUnu).x`. Programul va afişa la execuţie:

```
1 2 1 4
```

3.5.3. Conversiile implicite de la tipurile primitive la clasele *wrapper*

Pentru a uşura lucrul cu tipuri primitive, Java permite realizarea de conversii automate de la tipurile primitive la clasele *wrapper*. Trebuie avut grijă ca valorile utilizate să fie de tipul dorit. De exemplu, dacă vrem să folosim valoarea `5` (care este de tip `int`) ca un `float`, vom scrie `5f`, pentru `double` `5.0` sau `5d`, iar pentru `long` `5l`.

Construcţia:

```
Integer integer = 5;
```

este echivalentă cu:

```
Integer integer = new Integer(5);
```

Exemplul 3.5.7. Secvenţa de cod următoare foloseşte conversia implicită de la tipurile primitive la clasele *wrapper*.

```
Integer integer = 5;
Long l = 5l;
Float f = 5f;
Double d = 5.0;

Boolean b = true;
if (b.equals(true)) {
  System.out.println("E adevarat!");
}

Character ch = 'z';
if (ch.compareTo('a') > 0)
  System.out.println("OK");
```

Metoda `equals()` are un parametru de tip `Object`. Pentru că în exemplul de mai sus s-a specificat un tip primitiv (`boolean`), se va converti implicit la clasa *wrapper* (`Boolean`). Similar se realizează conversia caracterului `'a'` la clasa `Character`.

3.5.4. Conversiile implicite de la clasele *wrapper* la tipurile primitive

Tot pentru uşurinţa în utilizare se realizează conversia automată de la tipurile *wrapper* la cele primitive.

Exemplul 3.5.8. În programul de mai jos s-a folosit conversia implicită de la tipul `Integer` la `int`.

```
public class ConversiiLaPrimitive {

  public static int f(int x) {
    return 2*x+1;
  }

  public static void main(String[] args) {
    Integer x = 4;
    // x este convertit la int
    Integer fx = f(x);
    // x si fx sunt convertiti la int
    System.out.println(fx + x);
  }
}
```

Pentru că parametrul metodei `f()` este de tip `int` se realizează implicit conversia de la `Integer` la `int`. Operatorul + lucrează cu tipuri primitive sau cu `String`. În cazul prezentat mai sus se realizează conversia ambilor operanzi de la tipul `Integer` la `int`.

Execuţia programului va conduce la afişarea valorii `4`.

3.6. Concluzii

Capitolul 3 s-a oprit asupra prezentării tipului referinţă, mai precis a claselor, interfeţelor şi tablourilor. Secţiunea 3.2 descrie tipul clasă şi anume: sintaxă, domeniul de vizibilitate, modificatorii clasei, clase abstracte şi derivate, declaraţiile membrilor (atribute şi metode), supraîncărcare şi suprascriere, niveluri de acces, iniţializatori statici, declaraţiile constructorilor, clase interioare, clase generice, enumerări, distrugerea obiectelor şi eliberarea memoriei şi împărţirea unei aplicaţii în mai multe fişiere-sursă. Secţiunea 3.3 se referă la tipul interfaţă şi anume la sintaxă, atribute, metode, moştenire multiplă, ascundere, suprascriere şi supraîncărcare. Secţiunea 3.4 prezintă tablourile, acestora descriindu-li-se sintaxa, crearea, iniţializarea şi transmiterea către metode. Secţiunea 3.5 prezintă conversiile tipului referinţă (rămase „restante" din capitolul 2, unde s-au ilustrat conversiile tipurilor primitive). Sunt prezentate atât conversiile implicite/explicite, cât şi cele de compilare/execuţie. De asemenea, sunt prezentate conversiile implicite de la tipurile primitive la tipurile claselor *wrapper* şi invers, de la clasele *wrapper* la tipurile primitive.

Încheiem concluziile capitolului 3 cu asemănările şi deosebirile dintre interfeţe şi clase abstracte:

1. O clasă abstractă poate implementa una sau mai multe metode. O interfaţă nu implementează nici o metodă.
2. O clasă poate implementa mai mult de o interfaţă, dar poate extinde doar (cel mult) o clasă abstractă.
3. O interfaţă este utilizată de compilator chiar la compilare pentru verificarea tipurilor. Dimpotrivă, o clasă abstractă implică moştenire, în care determinarea metodei adecvate din mulţimea metodelor supraîncărcate se realizează la momentul execuţiei.
4. O clasă abstractă implică moştenirea metodelor deja definite şi definirea celor abstracte (în vederea creării de obiecte); o interfaţă specifică doar scheletul unei clase, fără posibilitatea moştenirii ulterioare.

3.7. Test-grilă

Întrebarea 3.7.1. Fie următorul program Java:

```
class C1 {
  int x = 1;
  C1() {
    System.out.print("x = " + x);
  }
}
class C2 extends C1 {
  int y = 3;
  C2(int y) {
    this.y = y;
  }
}
public class Test {
  public static void main(String[] args) {
    C2 obiect = new C2();
    System.out.println(" y = " + obiect.y);
  }
}
```

Ce puteţi spune despre acesta?

a) Eroare la compilare: nu există constructor fără argumente.
b) Programul este corect şi la execuţie va afişa: `y = 3`.
c) Programul este corect şi la execuţie va afişa: `x = 1 y = 3`.

Întrebarea 3.7.2. Fie următorul program Java:

```
class C0 {
  int x = 1;
}
class C1 {
  int y = 2;
}
class C2 extends C0, C1 {
  int z = 3;
}
public class TestDoi {
  public static void main(String[] args) {
    C2 obiect = new C2();
    System.out.println(obiect.x + obiect.y + obiect.z);
  }
}
```

Ce puteţi spune despre acesta?

a) Eroare la compilare: clasa `C2` nu poate fi derivată direct din două clase.
b) Eroare la compilare: clasa `C2` nu are constructor fără argumente.
c) Programul este corect şi la execuţie va afişa: `6`.

Întrebarea 3.7.3. Fie următorul program Java:

```
class x {
  private int x = 1;
  void x() {
    System.out.print(x + " ");
  }
}
class y extends x {
  private int x = 2;
  void x() {
    super.x();
    System.out.print(x);
  }
}
public class TestAtribute {
  public static void main(String[] args) {
    y obiect = new y();
    obiect.x();
  }
}
```

Ce puteţi spune despre acesta?

a) Eroare la compilare: nu se pot defini un atribut şi o metodă cu acelaşi nume (x).
b) Eroare la compilare: nu se poate defini un atribut cu numele clasei (x).

c) Eroare la compilare: nu se poate defini un atribut cu acelaşi nume într-o clasă derivată (x).
d) Programul este corect şi la execuţie va afişa: `1 2`.

Întrebarea 3.7.4. Ce se va afişa la execuţia următorului program?

```
public class TestStatic {
  public static void main(String args[]) {
    Exemplu a = new Exemplu();
    Exemplu b = new Exemplu();
    System.out.print("a.x = " + a.x);
    a.x = 100; b.x = 200;
    System.out.print(" a.x = " + a.x);
  }
}
class Exemplu {
  static int x = 0;
  Exemplu() {x++;};
}
```

a) `a.x = 2 a.x = 200`
b) `a.x = 0 a.x = 100`
c) `a.x = 1 a.x = 100`
d) `a.x = 1 a.x = 101`
e) `a.x = 2 a.x = 100`

Întrebarea 3.7.5. Ce puteţi spune despre următorul program?

```
public class Test {
  int i = 3, j = 4;
  public static void main(String [] arg) {
    System.out.println(i++ + ++j);
  }
}
```

a) Va apărea eroare la compilare, deoarece parametrul metodei `main()` trebuie să fie `String args []`.
b) Va apărea eroare la compilare, deoarece nu s-au inserat paranteze între operatorii ++, adică `(i++)` şi `(++j)`.
c) Va apărea eroare la compilare, deoarece din funcţia statică `main()` nu putem accesa variabilele nestatice `i` şi `j`.
d) La execuţie se va afişa `8`.
e) La execuţie se va afişa `9`.

Întrebarea 3.7.6. Ce puteţi afirma despre următorul program?

```
interface I1 { float x = 2.3f; }
public class Test implements I1 {
  public static void main(String[] args) {
    System.out.print(x + " ");
    x = 6.7f;
    System.out.println(x);
  }
}
```

a) Va apărea eroare la compilare, deoarece valoarea variabilei x nu se mai poate modifica.
b) La execuţie se va afişa 2.3f 6.7f.
c) La execuţie se va afişa 2.3f 2.3f.
d) La execuţie se va afişa 2.3 6.7.
e) La execuţie se va afişa 2.3 2.3.

Întrebarea 3.7.7. Ce puteţi spune despre următorul program Java?

```
interface I1 { int x = 2; }
interface I2 extends I1 { int y = 3; }
class C1 { int y = 4; }
public class C3 extends C1 implements I2 {
  public static void main(String args[]) {
    System.out.println("x = " + x + ", y =" + y);
  }
}
```

a) Va apărea eroare la compilare, deoarece atributul y este ambiguu pentru C1 şi I2.
b) La execuţie se va afişa x = 2, y = 3.
c) La execuţie se va afişa x = 2, y = 4.

Întrebarea 3.7.8. Ce puteţi afirma despre următorul program Java?

```
class C1 {
  public String f() {
    return this.getClass().getName();
  }
}
abstract class C2 extends C1 {
  int x = 2;
  public abstract String f();
}
class C3 extends C2 {
  int y = 3;
  public String f() {
    return super.f() + ", x = " + x + ", y = " + y;
  }
}
```

```
public class TestAbstract {
   public static void main(String args[]) {
      C3 obiect = new C3();
      System.out.println(obiect.f());
   }
}
```

a) Programul este corect și va afișa la execuție: `C3, x = 2, y = 3.`
b) Programul este corect și va afișa la execuție: `C1, x = 2, y = 3.`
c) Va apărea eroare la compilare, deoarece în clasa `C2` s-a suprascris metoda `f()` neabstractă cu o metodă abstractă.
d) Va apărea eroare la compilare, deoarece apelul `super.f()` din clasa `C3` se referă la o metodă abstractă.
e) Va apărea eroare la execuție, deoarece se intră într-o recursie infinită în metoda `f()` din clasa `C3`.

Întrebarea 3.7.9. Ce modificare trebuie făcută următorului program Java pentru a fi corect?

```
abstract class C1 {
   int x = 2;
   public final abstract String f();
}
final class C2 extends C1 {
   int y = 3;
   public String f() {
      return "x = " + x + ", y = " + y;
   }
}
public class TestFinal {
   public static void main(String args[]) {
      C2 obiect = new C2();
      System.out.println(obiect.f());
   }
}
```

a) Trebuie șters cuvântul `final` din definiția metodei `f()` a clasei `C1`.
b) Trebuie șters cuvântul `final` din definiția clasei `C2`.
c) Trebuie șters cuvântul `abstract` din definiția clasei `C1`.
d) Trebuie ștearsă inițializarea atributului x din declarația `int x = 2;` a clasei `C1`.
e) Programul este corect și la execuție va afișa `x = 2, y = 3.`

Întrebarea 3.7.10. Ce puteți spune despre următorul program Java?

```
class C1 {
   int x = 1;
   void f(int x) { this.x = x; }
```

```
  int getX_C1() { return x; }
}
class C2 extends C1 {
  float x = 5.0f;
  int f(int x) { super.f((int)x); }
  float getX_C2() { return x; }
}
public class SuprascriereSiAscundere {
  public static void main(String args[]) {
    C2 obiect = new C2();
    obiect.f(4);
    System.out.print(obiect.getX_C2() + " ");
    System.out.println(obiect.getX_C1());
  }
}
```

a) Programul este corect şi va afişa la execuţie: `5.0 4`.

b) Programul este corect şi va afişa la execuţie: `5 4`.

c) Programul este corect şi va afişa la execuţie: `4.0 4`.

d) Va apărea eroare la compilare, deoarece în clasa `C2` s-a suprascris greşit atributul `x` din clasa `C1`.

e) Va apărea eroare la compilare, deoarece metoda suprascrisă `f()` din clasa `C2` întoarce un tip diferit de `void`.

Întrebarea 3.7.11. Ce puteţi afirma despre următorul program Java?

```
class C1 {
  static String f() { return "Mesajul Unu din C1"; }
  String g() { return "Mesajul Doi din C1"; }
}
class C2 extends C1 {
  static String f() { return "Mesajul Unu din C2"; }
  String g() { return "Mesajul Doi din C2"; }
}
public class Test {
  public static void main(String[] args) {
    C1 obiect = new C2();
    System.out.println(obiect.f() + ", " + obiect.g());
  }
}
```

a) Programul este corect şi va afişa la execuţie: `Mesajul Unu din C1, Mesajul Doi din C1`.

b) Programul este corect şi va afişa la execuţie: `Mesajul Unu din C1, Mesajul Doi din C2`.

c) Programul este corect şi va afişa la execuţie: `Mesajul Unu din C2, Mesajul Doi din C1`.

d) Programul este corect şi va afişa la execuţie: `Mesajul Unu din C2, Mesajul Doi din C2.`
e) Va apărea eroare la compilare, deoarece în clasa `Test` variabila `obiect` **nu** aparţine clasei `C2`.

Întrebarea 3.7.12. Ce puteţi spune despre următorul program Java?

```
class C1 {
  double x = 5.0f;
  void f(int x) {
    f((double)x + 2);
  }
  void f(double x) {
    this.x = x;
  }
}
public class Supraincarcare {
  public static void main(String args[]) {
    C1 obiect = new C1();
    obiect.f(4.0);
    System.out.println(obiect.x);
  }
}
```

a) Programul este corect şi va afişa la execuţie: `4.0`.
b) Programul este corect şi va afişa la execuţie: `5.0`.
c) Programul este corect şi va afişa la execuţie: `6.0`.
d) Va apărea eroare la compilare, deoarece există ambiguitate în alegerea metodei `f()`.
e) Va apărea eroare la execuţie, deoarece metoda `f(int)` intră într-o recursie infinită.

Întrebarea 3.7.13. Cercetaţi corectitudinea următorului program Java:

```
class C1 {
  static int f() { return j; }
  static int i = f();
  static int j = 1;
}
public class TestStatic {
  public static void main(String[] args) {
    System.out.println(C1.i);
  }
}
```

a) Programul este corect şi va afişa la execuţie `0`.
b) Programul este corect şi va afişa la execuţie `1`.

c) Va apărea eroare la compilare, deoarece nu există nici un obiect declarat al clasei `C1`.
d) Va apărea eroare la compilare, deoarece membrii clasei `C1` nu sunt într-o ordine corectă.
e) Va apărea eroare la compilare, deoarece s-a omis declararea clasei `C1` ca fiind `static`.

Întrebarea 3.7.14. Ce puteţi afirma despre următorul program Java (liniile sunt numerotate)?

```
1. public class SubiectLicentaDoi {
2.   static int x = 10;
3.   static { x += 5; }
4.   public static void main(String args[]) {
5.     System.out.println("x = " + x);
6.   }
7.   static { x /= 5; }
8. }
```

a) Liniile 3 şi 7 nu se vor compila, deoarece lipsesc numele metodelor şi tipurile returnate.
b) Linia 7 nu se va compila, deoarece poate exista doar un iniţializator static.
c) Codul se compilează şi execuţia produce ieşirea `x = 10.`
d) Codul se compilează şi execuţia produce ieşirea `x = 15.`
e) Codul se compilează şi execuţia produce ieşirea `x = 3.`

Întrebarea 3.7.15. Ce se va afişa la execuţia următorului program Java?

```
public class SuprascriereSupraincarcare extends Baza {
  public void functie(int j) {System.out.print(" j = " + j);}
  public void functie(String s) {System.out.print(" s = " + s);}
  public static void main(String args[]) {
    Baza b1 = new Baza();
    Baza b2 = new SuprascriereSupraincarcare();
    b1.functie(5);
    b2.functie(6);
  }
}
class Baza {
  public void functie(int i) {System.out.print(" i = " + i);}
}
```

a) `i = 5 i = 6.`
b) `j = 5 j = 6.`
c) `i = 5 j = 6.`
d) `s = 5 s = 6.`

e) Eroare la compilare, deoarece definiţia clasei `Baza` apare după definiţia clasei `SuprascriereSupraincarcare`.
f) Eroare la compilare, deoarece lipseşte cuvântul `virtual` din metoda `functie()` a clasei `Baza`.

Întrebarea 3.7.16. Ce puteţi spune despre următorul program Java?

```
class C1 {
  static int x = 1;
  int y = 2;
  static C1() { x = 4; }
  C1(int y) {
    this.y = y + 1;
  }
}
public class Test {
  public static void main(String args[]) {
    System.out.print(C1.x + " ");
    C1 obiect = new C1(5);
    System.out.println(obiect.y);
  }
}
```

a) Programul este corect şi va afişa la execuţie: `1 6`.
b) Programul este corect şi va afişa la execuţie: `4 6`.
c) Programul este corect şi va afişa la execuţie: `1 2`.
d) Programul este corect şi va afişa la execuţie: `4 2`.
e) Va apărea eroare la compilare, deoarece s-a declarat unul dintre constructorii clasei `C1` ca fiind `static`.

Întrebarea 3.7.17. Ce puteţi afirma despre următorul program Java?

```
class C1 {
  int x = 1;
  C1(int x) { this.x = x; }
}
class C2 extends C1 {
  int y = 3;
  C2(int x) {
    this(x, y);
  }
  C2(int x, int y) {
    super(x);
    this.y = y;
  }
  public String toString() {
      return "x = " + x + ", y = " + y;
    }
}
public class Test {
  public static void main(String args[]) {
    C2 obiectUnu = new C2(4);
    C2 obiectDoi = new C2(5, 6);
    System.out.print(obiectUnu + " ");
    System.out.println(obiectDoi);
  }
}
```

a) Programul este corect şi va afişa la execuţie: `x = 4, y = 3 x = 5, y = 6.`
b) Programul este corect şi va afişa la execuţie: `x = 1, y = 3 x = 5, y = 6.`
c) Programul este corect şi va afişa la execuţie: `x = 1, y = 3 x = 1, y = 6.`
d) Programul este corect şi va afişa la execuţie: `x = 1, y = 3 x = 1, y = 3.`
e) Va apărea eroare la compilare, deoarece primul constructor al clasei `C2` conţine un apel explicit în care al doilea parametru este o variabilă instanţă.

Întrebarea 3.7.18. Indicaţi dacă programul următor este corect şi ceea ce se va afişa la execuţia acestuia:

```
class C1 {
  int x = 1;
  C1(int x) { this.x = x; }
}
class C2 extends C1 {
  int y = 2;
  public String toString() {
    return "x = " + x + ", y = " + y;
  }
}
public class Test {
  public static void main(String args[]) {
    C2 obiectUnu = new C2();
    C2 obiectDoi = new C2(3);
    System.out.print(obiectUnu);
    System.out.print(obiectDoi);
  }
}
```

a) Programul este corect şi va afişa la execuţie: `x = 1, y = 2 x = 1, y = 2.`
b) Programul este corect şi va afişa la execuţie: `x = 1, y = 2 x = 3, y = 2.`
c) Programul este corect şi va afişa la execuţie: `x = 1, y = 3 x = 1, y = 3.`
d) Programul este corect şi va afişa la execuţie: `x = 1, y = 3 x = 3, y = 2.`
e) Va apărea eroare la compilare, deoarece clasa `C1` nu are constructor fără parametri.

Întrebarea 3.7.19. Stabiliţi dacă programul următor este corect sau nu:

```
interface I1 {
  int j = 3;
  int [] i = {j + 1, j + 2};
}
interface I2 {
  int j = 5;
  int [] i = {j + 2, j + 4};
}
public class Test implements I1, I2 {
  public static void main(String[] args) {
    System.out.print(I1.j + " ");
    System.out.print(I2.i[1] + " ");
    System.out.print(I1.i[0]);
  }
}
```

a) Programul este corect şi va afişa la execuţie: `3 9 4`.
b) Va apărea eroare la compilare, deoarece interfeţele `I1` şi `I2` conţin declaraţiile aceloraşi atribute.
c) Va apărea eroare la compilare, deoarece în interfeţele `I1` şi `I2` atributul `i` conţine referinţă la atributul `j`.
d) Va apărea eroare la compilare, deoarece interfeţele `I1` şi `I2` conţin declaraţiile unui atribut de tip tablou.
e) Va apărea eroare la compilare, deoarece atributele interfeţelor `I1` şi `I2` nu se pot accesa dintr-o metodă statică.

Întrebarea 3.7.20. Ce puteţi spune despre corectitudinea următorului program Java?

```
interface I0 {
  int x;
}
interface I1 extends I0 {
  int [] y = {x + 1, x + 2};
}
interface I2 extends I0 {
  int [] y = {x + 3, x + 4};
}
interface I3 extends I1, I2 {
  int z = I1.y[0], t = I2.y[1];
}
public class TestAtribute implements I3 {
  public static void main(String[] args) {
    I0.x = 3;
    System.out.print(x + " ");
```

```
      System.out.print(z + " ");
      System.out.print(t);
    }
}
```

a) Programul este corect și va afișa la execuție: `3 4 7`.
b) Va apărea eroare la compilare, deoarece interfețele `I1` și `I2` conțin declarațiile acelorași atribute.
c) Va apărea eroare la compilare, deoarece în interfața `I0` atributul `x` nu este inițializat.
d) Va apărea eroare la compilare, deoarece interfața `I3` accesează un atribut ambiguu.
e) Va apărea eroare la compilare, deoarece atributele interfețelor `I0` și `I3` nu se pot accesa dintr-o metodă statică.

Întrebarea 3.7.21. Ce puteți afirma despre următorul program Java?

```
class ExceptiaNoastraUnu extends Exception {
  ExceptiaNoastraUnu() { super(); }
}
class ExceptiaNoastraDoi extends Exception {
  ExceptiaNoastraDoi() { super(); }
}
interface I1 {
  int f(int x) throws ExceptiaNoastraUnu, ExceptiaNoastraDoi;
}
interface I2 extends I1 {
  int f(int x) throws ExceptiaNoastraDoi;
}
class C1 implements I1 {
  public int f(int x) throws ExceptiaNoastraUnu,
  ExceptiaNoastraDoi {
    if (x < 0) throw new ExceptiaNoastraUnu();
    return x;
  }
  public int f(int x) throws ExceptiaNoastraDoi {
    if (x < 0) throw new ExceptiaNoastraDoi();
    return x;
  }
}
public class TestSuprascriereMetode {
  public static void main(String[] args) {
    C1 obiect = new C1();
    try {
      System.out.print(obiect.f(2) + " ");
      System.out.print(obiect.f(-2));
      }
      catch (ExceptiaNoastraUnu e) {
        System.out.print("A aparut o exceptie: " + e);
      }
      catch (ExceptiaNoastraDoi e) {
        System.out.print("A aparut o exceptie: " + e);
      }
    }
}
```

a) Programul este corect şi va afişa la execuţie: `2 A aparut o exceptie: ExceptiaNoastraUnu`.
b) Programul este corect şi va afişa la execuţie: `2 A aparut o exceptie: ExceptiaNoastraDoi`.
c) Va apărea eroare la compilare, deoarece metoda `f()` din interfaţa `I2` intră în contradicţie cu metoda `f()` din `I1`.
d) Va apărea eroare la compilare, deoarece interfeţele `I1` şi `I2` nu conţin atributul `x`.
e) Va apărea eroare la compilare, deoarece în clasa `C1` nu se poate implementa metoda `f()` din interfaţa `I1`.

Întrebarea 3.7.22. Selectaţi varianta corectă referitoare la corectitudinea programului Java de mai jos:

```
interface I1 {
  int f(int x);
}
interface I2 extends I1 {
  float f(float x);
  double f(double x);
}
class C1 implements I2 {
  public int f(int x) { return x; }
  public float f(float x) { return x; }
  public double f(double x) { return x; }
}
public class TestSupraincarcareMetode {
  public static void main(String[] args) {
    C1 obiect = new C1();
    System.out.print(obiect.f(2) + " ");
    System.out.print(obiect.f(4.5) + " ");
    System.out.print(obiect.f(2.5f));
  }
}
```

a) Programul este corect şi va afişa la execuţie: `2 4.5 2.5`.
b) Programul este corect şi va afişa la execuţie: `2 4.5 2.5f`.

c) Va apărea eroare la compilare, deoarece metodele `f()` din interfaţa `I2` intră în contradicţie cu metoda `f()` din `I1`.
d) Va apărea eroare la compilare, deoarece metodele `f()` din interfaţa `I2` intră în contradicţie între ele.
e) Va apărea eroare la compilare, deoarece metodele `f()` din interfeţele `I1` şi `I2` nu sunt declarate `public`.

Întrebarea 3.7.23. Cercetaţi corectitudinea următorului program:

```
public class Test {
  public static void main(String args[]) {
    C1 obiect = new C1();
    obiect.f(4, 3);
  }
}
class C1 {
  public void f(int xx, final int yy) {
    int a = xx + yy;
    final int b = xx - yy;
    class C2 {
      public void g() {
        System.out.print("a = " + a);
        System.out.print(", b = " + b);
      }
    }
    C2 obiectDoi = new C2();
    obiectDoi.g();
  }
}
```

a) Programul este corect şi va afişa la execuţie: `a = 7, b = 1`.
b) Programul este corect şi va afişa la execuţie: `a = 4, b = 3`.
c) Va apărea eroare la compilare, deoarece din metoda `g()` nu putem accesa variabila locală `a` din metoda `f()`.
d) Va apărea eroare la compilare, deoarece clasa `C2` nu poate fi definită în metoda `f()` din clasa `C1`.
e) Va apărea eroare la compilare, deoarece nu se creează în clasa `Test` un obiect de tip `C1.C2`.

Întrebarea 3.7.24. Verificaţi corectitudinea următorului program Java:

```
public class Exemplu {
  int v1[] = {0}, v2[] = {1};
  public static void main(String args[]) {
    int v1[] = {2}, v2[] = {3};
    System.out.print(v1[0] + " " + v2[0] + " ");
```

```
      f(v1, v2);
      System.out.println(v1[0] + " " + v2[0]);
   }
   public static void f(int v1[], int v2[]) {
      int w[] = {4};
      v1 = w;
      v2[0] = w[0];
      System.out.print(v1[0] + " " + v2[0] + " ");
   }
}
```

a) Programul este corect şi va afişa la execuţie: `0 1 4 4 0 1.`
b) Programul este corect şi va afişa la execuţie: `2 3 4 4 0 1.`
c) Programul este corect şi va afişa la execuţie: `2 3 4 4 2 4.`
d) Programul este corect şi va afişa la execuţie: `2 3 4 4 4 4.`
e) Va apărea eroare la compilare, deoarece atributele `v1` şi `v2` sunt redefinite ca variabile locale în metoda `main()`.

Întrebarea 3.7.25. Ce puteţi spune despre următorul program Java (liniile sunt numerotate)?

```
 1. class C1 { int x = 1; }
 2. class C2 extends C1 { int y = 2; }
 3. public class TestCompilare {
 4.    public static void main(String[] args) {
 5.       C1 obiectUnu = new C1();
 6.       System.out.print(obiectUnu.x + " ");
 7.       obiectUnu = new C2();
 8.       System.out.print(obiectUnu.x + " ");
 9.       C2 obiectDoi = (C2) obiectUnu;
10.       System.out.print(obiectDoi.x + " ");
11.    }
12. }
```

a) Programul este corect şi va afişa la execuţie: `1 1 1.`
b) Va apărea eroare la compilare la linia 7, deoarece `obiectUnu` este deja definit la linia 5.
c) Va apărea eroare la compilare la linia 9, deoarece `obiectUnu` este de tip `C1` definit la linia 5.
d) Vor apărea erori de compilare la liniile 5, 7 şi 9, deoarece într-o metodă statică nu putem crea variabile obiect.
e) Va apărea eroare la execuţie, aruncând excepţia `ClassCastException`.

Întrebarea 3.7.26. Stabiliţi care afirmaţii sunt adevărate pentru programul Java de mai jos (liniile sunt numerotate):

```
 1. public class TestTablouri {
```

```
 2.    public static void main(String[] args) {
 3.      int[] a = {2, 6};
 4.      Object b = a;
 5.      System.out.print(b[0] + " ");
 6.      byte[] c = new byte[2];
 7.      c = a;
 8.      System.out.print(c[1] + " ");
 9.    }
10.}
```

a) Programul este corect şi va afişa la execuţie: `2 6`.
b) Va apărea eroare la compilare doar la linia 4, deoarece `b` necesită o conversie explicită.
c) Va apărea eroare la compilare doar la linia 5, deoarece `b[0]` nu poate fi accesat.
d) Vor apărea erori de compilare la linia 5, deoarece `b[0]` nu poate fi accesat, şi la linia 7, deoarece tipurile componentelor tablourilor `a` şi `c` nu sunt compatibile.
e) Va apărea eroare la execuţie din cauza liniei 5, aruncându-se excepţia `ClassCastException`.

Întrebarea 3.7.27. Indicaţi dacă programul Java de mai jos este corect sau stabiliţi natura erorilor (liniile sunt numerotate):

```
 1. class C1 { int x = 1; }
 2. class C2 extends C1 { int y = 2; }
 3. interface I1 { int x = 3; void f(int x); }
 4. class C3 extends C1 implements I1 {
 5.   int x = 4;
 6.   public void f(int x) { this.x = x; }
 7. }
 8. public class TestConversii {
 9.   public static void main(String[] args) {
10.     C3 obiectUnu = new C3();
11.     I1 obiectDoi = obiectUnu;
12.     System.out.print(obiectDoi.x);
13.     C2 [] obiectTrei = new C2[3];
14.     System.out.print(obiectTrei[0].x + " ");
15.     C1 [] obiectPatru = obiectTrei;
16.     System.out.print(obiectPatru[1].x + " ");
17.     obiectTrei = (C2[]) obiectPatru;
18.   }
19. }
```

a) Programul este corect şi va afişa la execuţie: `3 2 1`.
b) Programul este corect la compilare şi va afişa la execuţie `3`, urmat de aruncarea excepţiei `NullPointerException` la linia 14.
c) Programul este corect la compilare şi va afişa la execuţie `3`, urmat de aruncarea excepţiei `NullPointerException` la linia 16.

d) Va apărea eroare la compilare doar la linia 15, deoarece tipurile componentelor tablourilor `obiectTrei` şi `obiectPatru` nu sunt compatibile.
e) Va apărea eroare la execuţie la linia 17, aruncându-se excepţia `ClassCastException`.

Întrebarea 3.7.28. Ce puteţi spune despre următorul program Java?

```
class C1 { int x = 1; }
interface I1 { int x = 2; void f(int x); }
class C2 extends C1 implements I1 {
  int x = 3;
  public void f(int x) { this.x = x; }
}

public class Test {
  public static void main(String[] args) {
    C1 obiectUnu = new C2();
    System.out.print(obiectUnu.x);
    I1 obiectDoi = (C2) obiectUnu;
    System.out.println(" " + obiectDoi.x);
  }
}
```

a) Programul este corect şi va afişa la execuţie: `1 2`.
b) Programul este corect şi va afişa la execuţie: `3 2`.
c) Programul este corect şi va afişa la execuţie: `3 3`.
d) Va apărea eroare de compilare la crearea lui `obiectDoi`.
e) Va apărea eroare la execuţie la afişarea lui `obiectDoi.x`, aruncându-se excepţia `ClassCastException`.

Întrebarea 3.7.29. Verificaţi dacă programul de mai jos este corect:

```
interface I1 {
  int x = 1;
  void f(int x);
}

class C1 implements I1 {
  int x = 2;
  public void f(int x) { this.x = x; }
}

public class Test {
  public static void main(String[] args) {
    C1 obiectUnu = new C1();
    System.out.print(obiectUnu.x);
```

```
      I1 obiectDoi = obiectUnu;
      System.out.println(" " + obiectDoi.x);
    }
}
```

a) Programul este corect şi va afişa la execuţie: `2 1`.
b) Programul este corect şi va afişa la execuţie: `2 2`.
c) Va apărea eroare de compilare la crearea lui `obiectDoi`, fiind necesară o conversie explicită de forma `(C2)`.
d) Va apărea eroare de compilare la crearea lui `obiectDoi`, deoarece atributul `x` se suprascrie.
e) Va apărea eroare la execuţie la afişarea lui `obiectDoi.x`, aruncându-se excepţia `ClassCastException`.

Întrebarea 3.7.30. Ce puteţi afirma despre următorul program Java?

```
class C1 {
  int x = 1;
  void f(int x) {
    System.out.print("Unu ");
    this.x = x;
  }
}

interface I1 {
  int x = 2;
  void f(int x);
}

class C2 extends C1 implements I1 {
  int x = 3;
  public void f(int x) {
    System.out.print("Doi ");
    this.x = x;
  }
}
public class Test {
  public static void main(String[] args) {
    C1 obiectUnu = new C2();
    System.out.print(obiectUnu.x + " ");
    obiectUnu.f(4);
    I1 obiectDoi = (C2) obiectUnu;
    obiectDoi.f(5);
    System.out.println(obiectDoi.x);
  }
}
```

a) Programul este corect şi va afişa la execuţie: `1 Unu Doi 2.`
b) Programul este corect şi va afişa la execuţie: `1 Unu Doi 3.`
c) Programul este corect şi va afişa la execuţie: `1 Doi Doi 2.`
d) Programul este corect şi va afişa la execuţie: `3 Doi Doi 2.`
e) Programul este corect şi va afişa la execuţie: `3 Doi Doi 3.`
f) Va apărea eroare de compilare la crearea lui `obiectDoi`, deoarece atributul x se suprascrie.
g) Va apărea eroare la execuţie la afişarea lui `obiectDoi.x`, aruncându-se excepţia `ClassCastException`.

Întrebarea 3.7.31. Indicaţi dacă programul de mai jos este corect şi, în caz afirmativ, ce va apărea pe ecran:

```
class C1 {
  int x = 1;
  void f(int x) {
    System.out.print("1 ");
    this.x = x;
  }
}
class C2 extends C1 {
  int x = 2;
  public void f(int x) {
    System.out.print("2 ");
    this.x = x;
  }
}
class C3 extends C1 {
  int x = 3;
  public void f(int x) {
    System.out.print("3 ");
    this.x = x;
  }
}
public class Test {
  public static void main(String[] args) {
    C1 obiectUnu = new C2();
    C1 obiectDoi = new C3();
    System.out.print(obiectUnu.x + " " + obiectDoi.x + " ");
    obiectUnu.f(4);
    obiectDoi.f(5);
    System.out.print(obiectUnu.x + " " + ((C2)obiectUnu).x +
    " ");
    System.out.print(obiectDoi.x + " " + ((C3)obiectDoi).x);
  }
}
```

a) Programul este corect şi va afişa la execuţie: `1 1 1 1 1 4 1 5.`
b) Programul este corect şi va afişa la execuţie: `1 1 2 3 1 4 1 5.`
c) Programul este corect şi va afişa la execuţie: `2 3 2 3 2 4 3 5.`
d) Programul este corect şi va afişa la execuţie: `2 3 2 3 4 4 5 5.`
e) Vor apărea erori de compilare la crearea instanţelor `obiectUnu` şi `obiectDoi`, deoarece atributul `x` se suprascrie.
f) Va apărea eroare la execuţie la afişarea lui `obiectUnu.x`, aruncându-se excepţia `ClassCastException`.

Întrebarea 3.7.32. Ce puteţi afirma despre programul Java de mai jos?

```
class C1 {
  int x = 1;
  C1 (int x) { this.x = x; }
}
class C2 extends C1 {
  int x = 3, y = 4;
  C2 (int x, int y) { super(x); this.y = y; }
}
public class Test {
  public static void main(String args[]) {
    C1[] tablouUnu = new C2[3];
    for (int i = 0; i < tablouUnu.length; i++)
      tablouUnu[i] = new C2(i * 2, i * 3);
    C2[] tablouDoi = (C2[]) tablouUnu;
    for (int i = 0; i < tablouUnu.length; i++)
     System.out.print(tablouDoi[i].x + " " + tablouDoi[i].y +
      " ");
  }
}
```

a) Programul este corect şi va afişa la execuţie: `1 0 1 3 1 6.`
b) Programul este corect şi va afişa la execuţie: `1 4 1 4 1 4.`
c) Programul este corect şi va afişa la execuţie: `3 0 3 3 3 6.`
d) Programul este corect şi va afişa la execuţie: `3 4 3 4 3 4.`
e) Vor apărea erori de compilare la crearea instanţelor `tablouUnu` şi `tablouDoi`, deoarece atributul `x` se suprascrie.
f) Va apărea eroare la execuţie la afişarea lui `tablouUnu.x`, aruncându-se excepţia `ClassCastException`.

Întrebarea 3.7.33. Verificaţi corectitudinea programului următor şi specificaţi ce se va afişa:

```
class C1 {
  int x = 1;
  C1 (int x) { this.x = x; }
}
```

```
class C2 extends C1 {
  int x = 2, y = 3;
  C2(int x) { super(x); }
  C2(int x, int y) { super(x); this.y = y; }
}
public class ApelExplicitConstructor {
  public static void main(String args[]) {
    C1 obiectUnu = new C1(4);
    C2 obiectDoi = new C2(5);
    C2 obiectTrei = new C2(6, 7);
    System.out.print(obiectUnu.x + " ");
    System.out.print(obiectDoi.x + " " + obiectDoi.y + " ");
    System.out.print(obiectTrei.x + " " + obiectTrei.y);
  }
}
```

a) Programul este corect şi va afişa la execuţie: `4 1 2 1 7`.
b) Programul este corect şi va afişa la execuţie: `4 1 3 6 7`.
c) Programul este corect şi va afişa la execuţie: `4 2 3 2 7`.
d) Programul este corect şi va afişa la execuţie: `4 5 3 6 7`.
e) Vor apărea erori de compilare la crearea instanţelor `obiectDoi` şi `obiectTrei`, deoarece atributul `x` se suprascrie.
f) Va apărea eroare la execuţie la afişarea lui `ObiectDoi.x`, aruncându-se excepţia `ClassCastException`.

3.8. Exerciţii propuse spre implementare

Exerciţiul 3.8.1. Scrieţi un program Java care descrie corpul numerelor complexe. Implementaţi constructori pentru iniţializarea unui număr complex şi metode pentru suma, produsul a două numere complexe şi modulul unui număr complex. Acesta poate avea scheletul:

```
class Complex {
  private double re, im;
  public Complex();
  public Complex(double re);
  public Complex(double re, double im);
  public Complex suma(Complex z);
  public Complex produs(Complex z);
  public double modul();
  public String toString();
}
```

Exerciţiul 3.8.2. Scrieţi un program Java pentru simularea operaţiunilor asupra unui cont bancar. Implementarea trebuie să conţină ca atribute numărul contului, tipul contului, numele şi adresa clientului, suma de bani existentă şi dobânda. Definiţi constructori cu parametri pentru iniţializarea numărului contului, a tipului contului, a numelui şi a adresei clientului. Ca metode, descrieţi depunerea unei sume de bani, retragerea unei sume de bani, aflarea sumei de bani curente (capital), aflarea dobânzii curente. Banca respectivă are două tipuri de conturi pentru persoane fizice: cu dobândă la vedere (cu posibilitate de operaţiuni zilnice) şi la termen (cu reţinerea sumei de bani până la termenul stabilit - se mai numeşte cont de economii). Suma de bani dintr-un cont la vedere poate chiar scădea (există posibilitatea dobânzii negative), dar nu şi suma de bani dintr-un cont la termen. La sfârşitul fiecărei luni/an, la suma iniţială se adaugă suma de bani obţinută din dobânda contului de la scadenţă. Folosind moştenirea, scrieţi clase Java pentru descrierea fiecărui tip de cont bancar. Utilizând un atribut static, număraţi câte persoane fizice au cont deschis la banca respectivă.

Exerciţiul 3.8.3. Scrieţi un program Java pentru lucrul cu figuri geometrice. Definiţi o clasă de bază `Punct` şi clasele derivate `Dreptunghi` şi `Cerc`. Testaţi aplicaţia creând câteva instanţe şi verificaţi metodele scrise. Cercul se reprezintă ca un punct (centrul cercului) şi o rază, iar dreptunghiul, ca două puncte diagonal opuse. Aplicaţia poate avea structura:

```
class Punct {
  protected int x, y;
  public Punct(int x, int y);
  public muta(int dx, int dy);
  public String toString();
}

class Dreptunghi extends Punct {
  protected int x0, y0;
  public Dreptunghi(int x, int y, int x0, int y0);
  public muta(int dx, int dy);
  public String toString();
}

class Cerc extends Punct {
  protected int r;
  public Cerc(int x, int y, int r);
  public muta(int dx, int dy);
  public String toString();
}
```

Exerciţiul 3.8.4. Scrieţi un program Java care defineşte o clasă `DataCalendaristica`. Datele calendaristice se pot trimite în două formate `ZZ/LL/AAAA` (formatul românesc de scriere a datelor calendaristice are forma `zi/lună/an`, de exemplu 20/12/2007) sau 20 decembrie 2007. Clasa `DataCalendaristica` are doi constructori care iniţializează ziua, luna şi anul datei calendaristice. Scrieţi şi câte o metodă de afişare a unui obiect din clasa `DataCalendaristica` (supraîncărcată) pentru fiecare dintre aceste formate de date calendaristice.

Exerciţiul 3.8.5. Implementaţi o clasă pentru numere raţionale. Aceasta va avea ca atribute numitorul şi numărătorul, trei constructori: implicit, cu un parametru şi cu doi. Se va scrie câte o metodă pentru operaţiile de adunare, scădere, înmulţire şi împărţire. Tot timpul fracţia reţinută de un obiect trebuie să fie ireductibilă (să nu se mai poată simplifica). Se va crea şi o clasă auxiliară pentru aflarea c.m.m.m.c şi c.m.m.d.c.

Exerciţiul 3.8.6. Scrieţi un program Java care să permită efectuarea de operaţii cu polinoame. Operaţiile permise vor fi adunarea, scăderea, înmulţirea, împărţirea, c.m.m.d.c, c.m.m.m.c. şi se va oferi posibilitatea aflării gradului unui polinom.

4. Şiruri şi structuri dinamice de date

4.1. Cuvinte-cheie

- şiruri de caractere, interfaţa `CharSequence`
- clasele `String`, `StringBuffer`, `StringTokenizer`
- structuri de date: colecţie, listă, vector, tabelă de asocieri, mulţime, stivă
- proprietăţi

4.2. Şiruri de caractere

Şirurile de caractere sunt frecvent utilizate în limbajele de programare pentru comunicarea dintre aplicaţia informatică şi utilizatorul uman sau între aplicaţiile informatice.

Pentru a lucra cu şiruri de caractere putem utiliza tablouri de caractere sau anumite interfeţe şi clase.

Tablourile au avantajul că fiecare caracter poate fi înlocuit foarte uşor. Uneori lungimea fixă a tabloului reprezintă un impediment. De altfel, în implementarea sa, clasa `String` foloseşte un tablou de caractere pentru stocarea şirului de caractere.

Interfaţa care modelează conceptul de *secvenţă de caractere* este `CharSequence`. Aceasta este implementată de clasele care desemnează o astfel de secvenţă: `CharBuffer`, `Segment`, `String`, `StringBuffer` şi `StringBuilder`. În acest subcapitol vom prezenta interfaţa `CharSequence` şi clasele `String`, `StringBuffer` şi `StringBuilder`.

4.2.1. Interfaţa **`CharSequence`**

Interfaţa `CharSequence` grupează o serie de metode pentru lucrul cu secvenţe de caractere. Prezentăm în tabelul de mai jos metodele definite de interfaţa `CharSequence`.

Prototipul metodei	Descrierea metodei
`char` **`charAt`**`(int index)`	Returnează caracterul de pe poziţia `index` din secvenţă. Numărătoarea caracterelor începe cu valoarea zero.
`int` **`length`**`()`	Întoarce numărul de caractere din secvenţă.
`CharSequence` **`subSequence`**`(int start, int end)`	Obţine subsecvenţa care începe de pe poziţia `start` şi se termină la poziţia end.
`String` **`toString`**`()`	Returnează şirul de caractere (`String`-ul) format de secvenţa de caractere.

Pentru metodele `charAt()` şi `subSequence()`, dacă una dintre poziţiile specificate nu se află în intervalul `0 ..' length()-1`, se va obţine excepţia `IndexOutOfBoundsException`, care este derivată din `RuntimeException`. Aşadar, nu este obligatoriu ca aceste metode să fie incluse într-un bloc `try-catch` sau metoda în care sunt utilizate să specifice în clauza `throws` că poate arunca această excepţie.

Aceste metode le vom regăsi în clasele `String`, `StringBuffer` şi `StringBuilder`, întrucât ele implementează interfaţa `CharSequence`. Aceste clase vor fi prezentate în secţiunile următoare.

4.2.2. Clasa `String`

Reamintim faptul că tipul `char` este un tip primitiv, care memorează un caracter reprezentat pe 16 biţi. Am văzut că putem defini şi lucra cu şiruri de caractere. Însă Java pune la dispoziţie tipul `String`, având astfel o manieră elegantă de lucru cu şirurile de caractere. `String`-urile sunt obiecte din clasa `String`.

Un literal `String` este o secvenţă de caractere cuprinsă între ghilimele (de exemplu: `"Acesta este un literal"`). Dacă se doreşte afişarea ghilimelelor, atunci se foloseşte `\"`, iar `\n` pentru trecerea la linie nouă ş.a.m.d.

Crearea unei variabile `String` se realizează exact ca şi în cazul celorlalte variabile.

Exemplul 4.2.1

```
String sir = "Primul sir";
sir = "Al doilea sir";
```

Când se atribuie unei variabile de tip `String` o altă valoare, vechea referinţă se pierde, valoarea sa urmând a fi eliberată automat din memorie de către colectorul de gunoaie. Se mai spune că obiectele `String` sunt fixe (nemodificabile).

Dacă un şir se iniţializează cu un literal `String`, atunci i se va atribui adresa acestuia din memorie. Literalii `String` sunt păstraţi în tabela de literali (engl. *literal pool*). Operatorul == aplicat pentru două `String`-uri testează dacă acestea pointează către aceeaşi adresă de memorie. În exemplul de mai jos, vom puncta această diferenţă sensibilă.

Exemplul 4.2.2

```
// literalul "abc" are o adresa fixa din memorie
String a1 = "abc";
// a1 si b1 pointeaza la aceeasi adresa de memorie
String b1 = "abc";

if (a1 == b1)
  System.out.println("1. Egalitate");
else
  System.out.println("1. Diferenta");

String a2 = "abc";
// b2 va pointa la alta adresa
// de memorie fata de a2
String b2 = new String("abc");

if (a2 == b2)
  System.out.println("2. Egalitate");
else
  System.out.println("2. Diferenta");
```

Execuţia secvenţei de instrucţiuni de mai sus va conduce la următorul afişaj la consolă:

```
1. Egalitate
2. Diferenta
```

Utilizarea operatorului `new` pentru un `String` conduce în momentul execuţiei codului la crearea unui spaţiu în memoria programului, şi nu în tabela de literali. Putem stabili ca un `String` creat cu `new` să fie plasat în tabela de literali pentru o reutilizare ulterioară sau pentru refolosirea unui `String` identic din tabela de literali. Astfel putem utiliza metoda `intern()` din clasa `String`. În programele care folosesc multe `String`-uri similare, acest procedeu poate reduce spaţiul de memorie utilizat. Mai mult, în programele care necesită comparaţii de `String`-uri memorate în tabela de literali, putem utiliza operatorul == (ce verifică egalitatea referinţelor), în loc de metoda `equals()` – mult mai lentă, deoarece verifică egalitatea caracter cu caracter.

Exemplul 4.2.3

```
String a1 = new String("abc");
String b1 = new String("abc");
String a2 = a1.intern(); // a2 se afla in tabela de literali
String b2 = b1.intern(); // b2 se afla in tabela de literali
// a2 si b2 pointeaza la aceeasi adresa de memorie
if (a2 == b2) // adevarat
System.out.println("Sirurile pointeaza la aceeasi adresa");
else
System.out.println("Sirurile pointeaza la adrese diferite");
```

Putem utiliza tablouri de `String`-uri.

Exemplul 4.2.4. Crearea tablourilor de `String`-uri.

```
String[] nume = new String[5];
nume[0] = "Goldan";
nume[1] = "Vasile";
```

Amintim şi faptul că metoda `main()` are ca argument un şir de `String`-uri, permiţând astfel primirea de argumente de la linia de comandă.

Operatorul + concatenează două (sau mai multe) şiruri de caractere. Operatorul += creează un nou `String`, adăugând şirul din partea dreaptă a operatorului.

Exemplul 4.2.5

```
String sir1 = "abc" + "def";
String sir2 = "abc";
sir2 += "def";
System.out.println("sir2 = " + sir2); // raspuns: sir2 = abcdef
```

În exemplul de mai sus, putem spune că `String`-urile sunt nemodificabile? Da. În realitate, se mai creează o zonă de memorie – necesară pentru memorarea noului cuvânt (`"abcdef"`) şi vechea valoarea (`"abc"`) se pierde.

Operatorul + se asociază de la stânga la dreapta. Dacă un operand este `String`, atunci se converteşte celălalt operand la `String`, tipul rezultatului întors fiind `String`.

Exemplul 4.2.6. Următoarea secvenţă de cod va afişa mesajul `cincizeci si cinci = 55 10 = zece`.

```
String sir3 = "cincizeci si cinci = " + 5 + 5;
String sir4 = 5 + 5 + " = zece ";
System.out.println(sir3 + " " + sir4);
```

Una dintre cele mai importante metode de conversie dintre tipuri arbitrare şi `String` este metoda statică `toString()`, care posedă numeroase variante supraîncărcate. Există, de asemenea, un număr de metode de convertire a şirurilor de caractere într-o varietate de tipuri.

Am văzut că tipurile predefinite `byte`, `short`, `int`, `float`, `double`, `long`, `boolean`, `char` nu necesită crearea instanţelor cu operatorul `new`. Pentru uniformizare, există clasele *wrapper* asociate tipurilor predefinite: `Integer`, `Float`, `Double`, `Long`, `Boolean`, `Character`. De exemplu, când se foloseşte operatorul + şi unul dintre operanzi este de tip `String`, iar celălalt de tip primitiv sau de clasă *wrapper* a unui tip primitiv, Java îl converteşte la `String` utilizând metoda statică `toString()` din clasele *wrapper*.

Exemplul 4.2.7

```
// conversie de la int la String:
int n = 123;
String s = Integer.toString(n); //sirul s devine "123"
// conversie de la float la String:
float f = 12.34f;
String s = Float.toString(f); // s devine "12.34"
// conversie de la String la int
String s = "1234";
int n = Integer.parseInt(s); // n devine 1234
// conversie de la String la float
String s = "12.34";
Float temp = Float.valueOf(s); // nu exista parseFloat
float f = temp.floatValue(); // f devine 12.34
```

Prezentăm mai jos câteva metode publice din clasa `java.lang.String`:

Prototipul metodei	Descrierea metodei
`boolean` **`equals`**`(Object unObiect)`	Compară două şiruri de caractere (obiectul apelat şi argumentul propriu-zis).

➔

➔

`boolean` **`equalsIgnoreCase`**`(String altSir)`	Se compară obiectul apelat (convertit la majuscule) şi argumentul `altSir` (convertit la majuscule) şi se returnează `true`, dacă sunt egale ca valoare, şi `false`, în caz contrar.
`int` **`compareTo`**`(String altSir)`	Returnează zero, dacă şirurile sunt egale, o valoare negativă, dacă şirul obiect precedă şirul parametru, şi o valoare pozitivă, dacă şirul obiect urmează şirului parametru în ordine lexicografică.
`String` **`replace`**`(char caracterVechi, char caracterNou)`	Înlocuieşte în şirul obiect orice apariţie a lui `caracterVechi` cu `caracterNou`.
`String` **`toLowerCase`**`()`	Converteşte orice majusculă a şirului obiect în minusculă.
`String` **`trim`**`()`	Şterge spaţiile albe de la capetele unui şir de caractere, inclusiv `\n` şi `\t`.
`int` **`length`**`()`	Întoarce numărul de caractere dintr-un şir obiect.
`String` **`substring`**`(int indexStart, int indexSfarsit)`	Extrage din şir partea specificată (subşirul care începe cu poziţia `indexStart` şi se termină cu poziţia `indexSfarsit`). Prima poziţie a unui şir este `0` şi ultimul caracter are indexul `length()-1`.
`char` **`charAt`**`(int index)`	Întoarce caracterul de la poziţia precizată de variabila `index`. Numărătoarea începe de la zero.
`int` **`indexOf`**`(String sir, int indexStart)`	Determină indexul de la care se găseşte `sir` în şirul obiect începând de la poziţia `indexStart`. Dacă nu este găsit, atunci se returnează `-1`.
`boolean` **`endsWith`**`(String sufix)`	Întoarce `true`, dacă `String`-ul `sufix` termină şirul obiect.
`char[]` **`toCharArray`**`()`	Returnează un tablou cu caracterele din şirul obiect.
`void` **`getChars`**`(int srcBegin, int srcEnd, char[] dst, int dstBegin)`	Se copiază caracterele din şirul obiect începând cu poziţia `srcBegin` până la poziţia `srcEnd` în tabloul de caractere `dst` începând cu poziţia `dstBegin`. În acest caz, trebuie să alocăm spaţiu pentru tabloul `dst`.
`byte[]` **`getBytes`**`()`	Caracterele sunt extrase într-un tablou de tip `byte`. Conversia dintre codul Unicode şi valorile pe 8 biţi (`byte`) va fi în concordanţă cu codificarea sistemului (octetul nesemnificativ de la caracterul Unicode va fi convertit în ASCII).

➔

`static String` **`copyValueOf`**`(char[] data)`	Această metodă statică creează un `String` din vectorul de caractere `data`.
`static String` **`copyValueOf`**`(char[] data, int offset, int count)`	Creează un `String` dintr-o submulţime de elemente ale unui tablou. Secvenţa de `count` caractere din şirul `data`, începând cu poziţia `offset`, va forma noul `String`.

Exemplul 4.2.8. Se compară caracter cu caracter şirurile de caractere `"Ion"` cu `"John"`, care, bineînţeles, că nu sunt egale:

```
String s = "Ion";
if (s.equals("John"))
  System.out.println("Ion este egal cu John");
```

Exemplul 4.2.9. La compararea şirurilor `"ion"` şi `"Ion"` se returnează `true`:

```
sir="ion";
if (sir.equalsIgnoreCase("Ion"))
  System.out.println("OK");
```

Exemplul 4.2.10. Şirul obiect este `"abc"`, iar cel parametru este `"ade"`. Variabila `n` va primi o valoare negativă.

```
int n = "abc".compareTo("ade"); //n este negativ
if (n == 0)
  rezultat = "sirurile sunt egale";
else
  if (n < 0)
    rezultat = "sir1 < sir2"; // Se va ajunge aici
  else
    rezultat = "sir1 > sir2";
```

Exemplul 4.2.11. Se va înlocui litera `m` din şirul `"mama"` cu litera `t` şi astfel va rezulta şirul `"tata"`.

```
sir = "mama".replace('m', 't'); // sir = "tata"
```

Exemplul 4.2.12. Se vor elimina spaţiile albe de la începutul şi sfârşitul `String`-ului `sir1`, iar rezultatul va fi atribuit lui `sir2`.

```
String sir1 = "   Un exemplu \n";
String sir2 = sir1.trim(); //sir2 devine "Un exemplu"
```

Exemplul 4.2.13. Obţinerea lungimii unui `String`:

```
int n = "Programarea in Java".length(); // n va fi 19
```

Reamintim că un tablou obişnuit avea o dată membră numită `length` (şi **nu** metodă).

Exemplul 4.2.14. Construirea unui subşir dintr-un şir:

```
sir1 = "pozitie";
sir2 = sir1.substring(2, 4); // sir2 = "zi"
```

Exemplul 4.2.15. Obţinerea caracterului de pe o anumită poziţie:

```
c = "pozitie".charAt(5); // c devine 'i'
c = "pozitie".charAt(1); // c devine 'o'
```

Exemplul 4.2.16. Determinarea poziţiei unui subşir:

```
int n = "abcdabcd".indexOf("bc",3); // n devine 5
```

Exemplul 4.2.17. Se verifică dacă un şir de caractere se termină cu un alt şir:

```
boolean e = "fisier.xml".endsWith(".xml"); // e va fi true
```

Exemplul 4.2.18. Dintr-un `String` putem obţine un tablou de caractere:

```
String text = "Convertim String in sir de caractere";
char [] sirCaractere = text.toCharArray();
System.out.println("" + sirCaractere[0] + sirCaractere[10]);
// afiseaza CS
```

Exemplul 4.2.19. Obţinerea unui subşir sub forma unui tablou de caractere:

```
String text = "Convertim subString in sir de caractere";
char [] sirCaractere = new char[3];
text.getChars(10, 13, sirCaractere, 0);
System.out.println("" + sirCaractere[0] + sirCaractere[1]
  + sirCaractere[2]);
// se va tipari "sub"
```

Exemplul 4.2.20. Obţinerea unui tablou de octeţi dintr-un şir de caractere:

```
String text = "Convertim String in sir de caractere";
byte[] sirCaractere = text.getBytes();
```

Exemplul 4.2.21. Secvenţa de cod Java va afişa textul `"Un String"`:

```
char[] textVector = {'U', 'n', ' ', 'S', 't', 'r', 'i', 'n', 'g'};
```

```
String text = String.copyValueOf(textVector);
System.out.println(text);
```

Exemplul 4.2.22. Următorul cod Java va conduce la afişarea cuvântului `String`:

```
char[] textVector = {'U', 'n', ' ', 'S', 't', 'r', 'i', 'n', 'g'};
String text = String.copyValueOf(textVector, 3, 6);
System.out.println(text); // va fi afisat "String"
```

4.2.3. Clasa `StringBuffer`

O altă clasă care se referă la procesarea intensivă a şirurilor de caractere este `StringBuffer`. Ea se ocupă de procesarea şirurilor de caractere de lungime variabilă. O instanţă a clasei `StringBuffer` reprezintă un `String` care poate fi modificat dinamic. Am văzut că un obiect `String` nu poate fi modificat, dar am creat obiecte noi de tip `String` obţinute prin modificarea obiectelor `String` existente. Şirurile din clasa `StringBuffer` îşi pot schimba conţinutul sau/şi lungimea prin alocarea unui spaţiu suplimentar de memorie.

Un obiect `StringBuffer` conţine un bloc de memorie numit buffer care poate sau nu să conţină un `String`, şi, dacă da, nu este obligatoriu să fie ocupat în totalitate. Aşadar, lungimea unui şir ţinut de un obiect `StringBuffer` poate fi diferită de lungimea bufferului (lungimea bufferului se numeşte capacitatea obiectului). Pentru a verifica valoarea capacităţii unui obiect `StringBuffer`, putem apela metoda `capacity()` din clasa `StringBuffer`.

Crearea unui obiect din clasa `StringBuffer` se realizează în mod obişnuit apelând unul dintre constructorii:

- `StringBuffer()`
- `StringBuffer(int capacitate)`
- `StringBuffer(String sir)`
- `StringBuffer(CharSequence seq)`

Primul constructor creează un şir cu capacitatea de 16 caractere, iar al doilea cu capacitatea specificată. Al treilea constructor permite iniţializarea şirului de caractere, iar capacitatea va fi egală cu lungimea şirului iniţial plus 16. Ultimul constructor permite crearea unui şir din obiectul care implementează interfaţa `CharSequence`.

Exemplul 4.2.23. Crearea unui obiect `StringBuffer`:

```
StringBuffer a = new StringBuffer(
   "Un prim exemplu de String dinamic");
```

Prezentăm în tabelul de mai jos câteva dintre metodele publice ale clasei `StringBuffer`.

Prototipul metodei	Descrierea metodei
`int capacity()`	Returnează capacitatea bufferului.
`StringBuffer delete(int start, int end)`	Şterge caracterele cuprinse între poziţiile `start` şi `end`.
`StringBuffer deleteCharAt(int index)`	Şterge caracterul de pe poziţia specificată.
`void ensureCapacity(int minimumCapacity)`	Măreşte capacitatea bufferului, astfel încât capacitatea finală să fie mai mare sau egală cu capacitatea specificată (`minimumCapacity`). Capacitatea va fi maximum dintre dublul capacităţii curente plus doi şi capacitatea specificată.
`void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)`	Caracterele cuprinse între poziţiile `srcBegin` şi `srcEnd` sunt copiate în tabloul `dst`, începând cu poziţia `dstBegin`.
`int indexOf(String str)`	Returnează prima poziţie în care apare şirul `str` în şirul de caractere ţinut în obiectul curent.
`int indexOf(String str, int fromIndex)`	Returnează prima poziţie, de după `fromIndex`, în care apare şirul `str` în şirul de caractere ţinut în obiectul curent.
`int lastIndexOf(String str)`	Returnează ultima poziţie în care apare şirul `str` în şirul de caractere ţinut în obiectul curent.
`int lastIndexOf(String str, int fromIndex)`	Returnează ultima poziţie, de după `fromIndex`, în care apare şirul `str` în şirul de caractere ţinut în obiectul curent.
`int length()`	Returnează lungimea şirului de caractere.
`StringBuffer replace(int start, int end, String str)`	Înlocuieşte caracterele cuprinse între poziţiile `start` şi `end` cu cele conţinute de şirul `str`.
`StringBuffer reverse()`	Inversează ordinea caracterelor în şir.
`void setCharAt(int index, char ch)`	Înlocuieşte caracterul de pe poziţia `index` cu caracterul dat de `ch`.

➔

➔

`void` **`setLength`**`(int newLength)`	Se modifică lungimea şirului de caractere curent. Când creştem lungimea, caracterele în plus vor fi iniţializate cu `'\u0000'`. Când descreştem lungimea, caracterele de la sfârşitul cuvântului se vor trunchia. Metoda nu afectează capacitatea bufferului decât dacă lungimea nouă este mai mare decât capacitatea. În acest caz, noua capacitate va fi dublul vechii capacităţi la care se adaugă 2. Dacă totuşi noua lungime depăşeşte capacitatea, atunci aceasta devine egală cu noua lungime.
`String` **`substring`**`(int start)`	Obţine subşirul care începe cu poziţia `start` şi ţine până la sfârşitul şirului.
`String` **`substring`**`(int start, int end)`	Obţine subşirul care începe cu poziţia `start` şi se termină la poziţia `end`.
`String` **`toString`**`()`	Returnează `string`-ul format de caracterele ţinute în obiectul curent.
`void` **`trimToSize`**`()`	Reduce capacitatea bufferului la lungimea efectivă a şirului de caractere stocat.

Exemplul 4.2.24. Aflarea lungimii unui şir şi a capacităţii `StringBuffer`-ului:

```
StringBuffer a = new StringBuffer("abcdef");
int lungime1 = a.length();          // va fi 6
int capacitate1 = a.capacity();    // va fi 22
StringBuffer b = new StringBuffer();
int lungime2 = b.length();          // va fi 0
int capacitate2 = b.capacity();    // va fi 16
```

Exemplul 4.2.25. Implicit, capacitatea unui obiect `StringBuffer` este lungimea obiectului `StringBuffer` la care se adună 16. Metoda `ensureCapacity()` permite să modificăm valoarea implicită a capacităţii unui obiect `StringBuffer`.

```
StringBuffer a = new StringBuffer("0123456789");
int lungime = a.length();          // va fi 10
int capacitate = a.capacity();    // va fi 26
a.ensureCapacity(30);
lungime = a.length();          // va fi 10
capacitate = a.capacity();    // va fi 54
```

Exemplul 4.2.26. Restrângerea lungimii unui şir va conduce la trunchierea acestuia:

```
StringBuffer a = new StringBuffer("0123456789");
a.setLength(8); // a va fi "01234567"
```

Metoda `append()` permite adăugarea unui şir de caractere la sfârşitul unui obiect `StringBuffer` existent. Lungimea şirului conţinut de obiectul `StringBuffer` va fi mărită (automat) cu lungimea şirului adăugat. Metoda `append()` returnează obiectul `StringBuffer` extins, deci îl putem asigna unui alt obiect `StringBuffer`.

Exemplul 4.2.27

```
StringBuffer a = new StringBuffer("abc");
StringBuffer b = a.append("def"); // a si b devin "abcdef"
// de fapt, a si b vor pointa asupra aceleiasi zone de memorie
b.setLength(4);
System.out.println(a); // a va fi "abcd"
```

O altă versiune de `append()` are încă doi parametri care oferă posibilitatea adăugării unui subşir de caractere la unul dat. Aceşti doi parametri suplimentari sunt indexul primului caracter care trebuie adăugat şi numărul total de caractere ce trebuie adăugate.

Exemplul 4.2.28. Din `String`-ul `b` se va adăuga şirului `a` subşirul `"fg"`.

```
StringBuffer a = new StringBuffer("abc");
String b = "defghi";
a.append(b, 2, 2); // a va deveni "abcfg"
```

Se pot adăuga la un obiect `StringBuffer` orice variabile din tipurile de bază: `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, `double`. În fiecare caz, valoarea este convertită la un `String` echivalent cu valoarea care se adaugă la obiect.

Exemplul 4.2.29

```
StringBuffer a = new StringBuffer("abc");
long x = 99;
a.append(x); // a devine "abc99"
char [] text = {'d', 'e', 'f', 'g', 'h', 'i'};
a.append(text, 2, 3); // a devine "abc99fgh"
```

Putem, de asemenea, insera `String`-uri (de fapt, valori de variabile din tipurile de bază) în orice poziţie a unui obiect `StringBuffer` (nu neapărat la sfârşitul acestuia) prin intermediul metodei `insert()`. În clasa `StringBuffer` există mai multe forme de inserare, dintre care amintim:

```
void insert(int index, String sir);
void insert(int index, char[] str, int pozitie, int lungime);
```

Similar ca la metoda `append()` se pot insera valori ale tipurilor primitive şi secvenţe de caractere.

Exemplul 4.2.30. Se va insera şirul `"def"` în şirul `"abc"` pe poziţia 1:

```
StringBuffer a = new StringBuffer("abc");
```

```
a.insert(1, "def"); // a va defini "adefbc"
```

Exemplul 4.2.31. Se va înlocui caracterul de pe poziţia 1 (`'b'`) cu caracterul `'d'`:

```
StringBuffer a = new StringBuffer("abc");
a.setCharAt(1, 'd'); // a va defini "adc"
```

Exemplul 4.2.32. Variabila `b` va conţine şirul `"abc def"`:

```
StringBuffer a = new StringBuffer("abc def");
String b = a.toString();
```

Metoda `toString()` este intensiv folosită de compilator împreună cu metoda `append()` pentru a implementa concatenarea obiectelor `String`.

Exemplul 4.2.33. Instrucţiunea:

```
String a = "abc" + "def";
```

va fi interpretată de compilator drept:

```
String a =
   new StringBuffer().append("abc").append("def").toString();
```

Exemplul 4.2.34. Inversarea ordinii caracterelor:

```
StringBuffer sb = new StringBuffer("zero");
sb.reverse();
System.out.println(sb); // se afiseaza "orez"
```

Exemplul 4.2.35. Reducerea capacităţii bufferului.

```
StringBuffer sb = new StringBuffer("crocodil");
System.out.println(sb.capacity()); // afiseaza 24
sb.trimToSize();
System.out.println(sb.capacity()); // afiseaza 8
```

Clasa `StringBuffer` poate fi folosită fără probleme de mai multe fire de execuţie. Eventual, atunci când se utilizează un singur fir de execuţie se poate folosi clasa `StringBuilder` care este similară cu `StringBuffer`. Clasa `StringBuilder` nu conţine elemente de sincronizare a firelor de execuţie şi, de aceea, metodele acesteia sunt executate mai rapid.

4.2.4. Clasa `StringTokenizer`

Clasa `java.util.StringTokenizer` este utilă pentru împărţirea unui şir într-o secvenţă de atomi lexicali (engl. *tokens*). Aceştia sunt definiţi ca o mulţime de caractere de delimitare. Delimitatorii obişnuiţi sunt: spaţiile, taburile, semnele de punctuaţie (`, . ;`).

Declararea unei variabile din această clasă se face similar cu alte variabile. Există trei constructori frecvent utilizaţi:

- `public StringTokenizer(String str);`
- `public StringTokenizer(String str, String delim);`
- `public StringTokenizer(String str, String delim, boolean returnDelims);`

Primul constructor creează un obiect din clasa `StringTokenizer`, care împarte şirul dat utilizând caracterele de delimitare standard (spaţiu, `'\r'`, `'\t'`, `'\n'`). Al doilea constructor creează un obiect din clasa `StringTokenizer`, care împarte şirul dat folosind caracterele din şirul `delim`. Dacă valoarea parametrului `returnDelims` este `true`, atunci delimitatorii sunt consideraţi ca token-uri şi fiecare delimitator este returnat ca un string de lungime unu.

Exemplul 4.2.36. Instanţierea unui obiect de clasă `StringTokenizer`:

```
StringTokenizer st = new StringTokenizer("abc de/ c"," /");
```

Metodele publice cele mai importante din clasa `StringTokenizer` sunt prezentate în tabelul de mai jos:

Prototipul metodei	Descrierea metodei
`String` **`nextToken`**`()`	Întoarce următorul atom lexical.
`String` **`nextToken`**`(String delim)`	Setează delimitatorii ca fiind caracterele din şirul dat şi returnează următorul atom lexical.
`boolean` **`hasMoreTokens`**`()`	Returnează `true`, dacă mai există date care pot forma un atom lexical.
`int` **`countTokens`**`()`	Calculează numărul de atomi lexicali care pot fi determinaţi.

Exemplul 4.2.37. Un prim exemplu de divizare a unui şir de caractere:

```
String data ="22 / 06 / 12 : 30";
String zi, luna, ora, min;
StringTokenizer d = new StringTokenizer(data," /");
zi = d.nextToken(); // zi = "22"
luna = d.nextToken();  // luna = "06"
ora = d.nextToken();  // ora  = "12"
min = d.nextToken(" :"); // min  = "30"
```

Exemplul 4.2.38. Bucla se va executa atât timp cât mai există *token*-uri:

```
String exemplu = "4,6 ,7, 10, 15 ,20";
String element;
```

```
StringTokenizer listaNumere = new StringTokenizer(exemplu," ,");
while (listaNumere.hasMoreTokens()) {
   element = listaNumere.nextToken();
      . . .
}
```

4.3. Structuri dinamice de date

În această secţiune vom prezenta structurile dinamice de date: lista, arborele binar, clasele `ArrayList`, `Vector`, `HashMap`, `Hashtable`, `Stack`, `BitSet`, `Properties` şi `HashSet`.

Pentru început vom descrie câteva interfeţe frecvent utilizate în Java: `Enumeration`, `Iterable`, `Iterator`, `Collection`, `Comparable`, `Comparator`, `Map`, `List`, `Set`.

4.3.1. Interfaţa `Enumeration`

Există colecţii de date ale căror elemente pot fi enumerate. Pentru acestea se utilizează interfaţa `Enumeration`. Aceasta permite un mod unitar de lucru cu mulţimi de elemente care pot fi enumerate.

Interfaţa `Enumeration` a fost definită folosind un tip de date generic *E*, care reprezintă tipul elementelor ce pot fi enumerate. Atunci când dorim să folosim această interfaţă, vom specifica tipul folosit. Dacă vrem să enumerăm elemente de tipuri diferite putem folosi clasă `Object`.

Interfaţa `Enumeration` conţine două metode:

Prototipul metodei	Descrierea metodei
`boolean` **`hasMoreElements`**`()`	Returnează o valoare booleană care indică dacă mai sunt elemente de enumerat.
`E` **`nextElement`**`()`	Returnează următorul element din enumerare.

Exemplul 4.3.1. Definim o clasă care implementează interfaţa `Enumeration`. Aceasta va conţine o mulţime de trei elemente de tip `String`.

```
public class Tripleta implements Enumeration<String> {
   private String[] elemente;
   private int index;

   public Tripleta() {
     elemente = new String[]{"primul", "al doilea", "al treilea"};
      index = 0;
```

```
    }

    public boolean hasMoreElements() {
      return index < elemente.length;
    }

    public String nextElement() {
      return elemente[index++];
    }
}
```

Observăm că sunt definite doar metodele din interfaţa `Enumeration`. Clasa definită mai sus poate fi utilizată astfel:

```
Enumeration enumeratie = new Tripleta();
while (enumeratie.hasMoreElements()) {
  System.out.println(enumeratie.nextElement());
}
```

De regulă, metodele `hasMoreElements()` şi `nextElement()` sunt utilizate în tandem. Adică se apelează întâi `hasMoreElements()` şi abia apoi `nextElement()`. De asemenea, nu se apelează `nextElement()` fără să se fi apelat în prealabil `hasMoreElements()`.

4.3.2. Iteratori

Iteratorii sunt obiecte care permit parcurgerea secvenţială a unei colecţii de date. Se aseamănă cu cele care implementează interfaţa `Enumeration`. În plus, iteratorii pot elimina elemente din colecţie.

Pentru iteratori s-a definit interfaţa `Iterator`. Aceasta foloseşte un tip generic *`E`* pentru tipul elementelor care sunt parcurse şi are trei metode:

Prototipul metodei	Descrierea metodei
`boolean` **`hasNext`**`()`	Returnează `true`, dacă mai sunt elemente de parcurs.
`E` **`next`**`()`	Întoarce următorul element. Dacă nu mai sunt elemente se va arunca o excepţie de tipul `NoSuchElementException`, care este derivată din `RuntimeException`.
`void` **`remove`**`()`	Elimină din colecţia de elemente pe ultimul returnat de iterator. Dacă operaţiunea de eliminare a elementelor nu este disponibilă se va arunca excepţia `UnsupportedOperationException`. Dacă nu a fost returnat nici un element de către iterator se va arunca `IllegalStateException`. Ambele excepţii sunt derivate din `RuntimeException`.

Interfaţa `Iterable` este utilizată pentru a specifica faptul că o colecţie de elemente are iteratori. Singura metodă este `interator()` care nu are parametri şi returnează un iterator, prototipul acestei metode fiind `Iterator<E> iterator()`. Întrucât interfaţa `Iterator` foloseşte un tip generic, interfaţa `Iterable` va utiliza acelaşi tip generic.

4.3.3. Colecţii

Interfaţa `Collection` modelează conceptul de colecţie de elemente. Aceasta este derivată din interfaţa `Iterable`. Prin urmare va folosi un tip generic pentru a desemna tipul elementelor din colecţie.

În general, elementele unei colecţii se pot repeta. Implementările particulare pot restricţiona acest lucru.

Metodele definite de interfaţa `Collection` sunt prezentate în tabelul de mai jos.

Prototipul metodei	Descrierea metodei
`boolean` **`add`**`(E e)`	Adaugă elementul `e` în colecţie. Returnează `true`, dacă s-a modificat colecţia, iar în caz contrar, se returnează `false`.
`boolean` **`addAll`**`(? extends E c)`	Adaugă toate elementele din colecţia `c` la colecţia curentă. Returnează `true`, dacă s-a modificat colecţia.
`void` **`clear`**`()`	Elimină toate elementele din colecţie.
`boolean` **`contains`**`(Object o)`	Verifică dacă obiectul specificat face parte din colecţie.
`boolean` **`containsAll`**`(Collection <?> c)`	Verifică dacă elementele din colecţia `c` fac parte şi din colecţia curentă.
`boolean` **`equals`** `(Object o)`	Verifică dacă două colecţii sunt egale.
`int` **`hashCode`**`()`	Returnează codul hash pentru obiectul curent.
`boolean` **`isEmpty`**`()`	Returnează `true`, dacă nu există elemente în colecţie.
`Iterator<E>` **`iterator`**`()`	Returnează un iterator pentru colecţia curentă.
`boolean` **`remove`** `(Object o)`	Elimină obiectul o singură dată din colecţie, dacă acesta există. Se returnează `true`, dacă s-a realizat eliminarea unui element.
`boolean` **`removeAll`**`(Collection <?> c)`	Elimină din colecţia `c` elementele care se găsesc în colecţia curentă. Returnează `true`, dacă s-a modificat colecţia.
`boolean` **`retainAll`**`(Collection <?> c)`	Reţine în colecţia curentă doar elementele care se regăsesc şi în colecţia `c`.

→

➔

int **size**()	Returnează numărul elementelor din colecţie.
Object[] **toArray**()	Obţine un tablou de obiecte cu elementele colecţiei.
T[] **toArray**(T[] a)	Returnează un tablou de tipul T, dacă elementele din colecţie pot fi convertite la tipul T. Dacă elementele colecţiei încap în tabloul a, atunci vor fi stocate în acesta şi va fi returnat. Altfel se va aloca un nou tablou, care va fi cel returnat.

4.3.4. Tabele de asocieri

Interfaţa Map este utilizată pentru colecţii de perechi de obiecte. Unui obiect numit *cheie* i se asociază un obiect numit *valoare*. Astfel, interfaţa Map va fi un tabel de asocieri cheie-valoare.

O cheie nu poate apărea decât o singură dată în tabela de asocieri. O valoare poate fi asociată mai multor chei.

Prezentăm în tabelul următor metodele interfeţei Map.

Prototipul metodei	Descrierea metodei
void **clear**()	Elimină toate asocierile.
boolean **containsKey**(Object key)	Verifică dacă este folosită cheia specificată.
boolean **containsValue**(Object value)	Verifică dacă valoarea specificată există în colecţia de asocieri.
Set<Map.Entry<K,V>> **entrySet**()	Returnează o mulţime (engl. *set*) de asocieri.
boolean **equals**(Object o)	Compară egalitatea obiectului curent cu cel specificat.
V **get**(Object key)	Returnează valoarea asociată cheii key.
int **hashCode**()	Obţine codul *hash* pentru obiectul curent.
boolean **isEmpty**()	Returnează true, dacă nu există asocieri.
Set<K> **keySet**()	Returnează mulţimea cheilor folosite.
V **put**(K key, V value)	Asociază valoarea value cheii key. Întoarce vechea valoarea asociată sau valoarea null, dacă nu exista vreo asociere pentru cheia respectivă.
void **putAll**(Map<? extends K, ? extends V> m)	Adaugă în tabela curentă toate asocierile din tabela m.

➔

➔

V **remove**(Object key)	Elimină asocierea pentru cheia specificată. Se returnează valoarea din vechea asociere.
int **size**()	Obţine numărul asocierilor din tabela curentă.
Collection<V> **values**()	Obţine o colecţie cu toate valorile din tabelă.

Interfaţa `Map.Entry` este utilizată pentru stocarea unei asocieri dintre o cheie şi o valoare. Aceasta conţine două tipuri generice: `K` pentru tipul cheilor şi `V` pentru cel al valorilor. Metodele specificate în această interfaţă sunt descrise în tabelul de mai jos.

Prototipul metodei	Descrierea metodei
boolean **equals**(Object o)	Testează egalitatea dintre obiectul curent şi cel transmis ca parametru.
K **getKey**()	Obţine cheia din asociere.
V **getValue**()	Obţine valoarea din asociere.
int **hashCode**()	Obţine codul hash pentru obiectul curent.
V **setValue**(V value)	Înlocuieşte vechea valoare asociată cu cea transmisă ca parametru. Se returnează vechea valoare asociată.

Cele mai utilizate clase care implementează interfaţa `Map` sunt `HashMap` şi `Hashtable`. Implicit, nu este bine a se *utiliza* `HashMap` de către mai multe fire de execuţie întrucât nu este sincronizată. În schimb, obiectele de tip `Hashtable` pot fi *utilizate* fără probleme de mai multe fire de execuţie.

Exemplul 4.3.2. Utilizarea clasei `Hashtable`. Pentru o valoare numerică reprezentând un identificator unic al unui angajat (marca) se asociază un obiect corespunzător angajatului.

```
import java.util.Hashtable;

class Angajat {
   String nume, prenume;

   public Angajat(String nume, String prenume) {
      this.nume = nume;
      this.prenume = prenume;
   }

   public String toString() {
      return prenume + " " + nume;
   }
}
```

```
public class TestHashtable {

   public static void main(String[] args) {
      Hashtable<Integer, Angajat> tabel =
         new Hashtable<Integer, Angajat>();

      tabel.put(3, new Angajat("Ionescu", "Vlad"));
      tabel.put(5, new Angajat("Zamfir", "Bogdan"));
      tabel.put(9, new Angajat("Golescu", "Nicu"));

      Angajat angajat = tabel.get(5);
      System.out.println(angajat);
   }

}
```

În urma execuţiei programului se va afişa:

```
Bogdan Zamfir
```

Exemplul 4.3.3. Folosim o tabelă cu scopul asocierii unei valori numerice pentru o serie de articole. Această tabelă poate fi utilizată pentru reprezentarea unei comenzi.

```
import java.util.HashMap;
import java.util.Iterator;

class Articol {
   String denumire;
   float pret;

   public Articol(String denumire, float pret) {
      this.denumire = denumire;
      this.pret = pret;
   }

   public String toString() {
      return denumire + " (" + pret + ")";
   }
}

public class TestHashMap {

   public static void main(String[] args) {
      // construim tabela de asocieri
      HashMap<Articol, Integer> articoleComandate =
        new HashMap<Articol, Integer>();

    // definim articolele
    Articol[] articole = new Articol[] {
        new Articol("Stilou", 12.50f),
        new Articol("Creion", 1f),
        new Articol("Radiera", 0.5f),
        new Articol("Caiet", 12f)
    };

    // completam tabela de asocieri
    articoleComandate.put(articole[1], 3);
    articoleComandate.put(articole[2], 1);
    articoleComandate.put(articole[3], 0);

    float total = 0;
    Integer cantitate;
    Iterator<Articol> it = articoleComandate.keySet().iterator();
    Articol articol;

    // parcurgem multimea cheilor
    while(it.hasNext()) {
      articol = it.next();
      cantitate = articoleComandate.get(articol);
      // daca produsul nu a fost comandat
      if (cantitate == null) continue;
      // daca produsul a fost comandat si cantitatea este
      // strict pozitiva
      if (cantitate > 0) {
        System.out.println(cantitate + " x " + articol);
        total += cantitate * articol.pret;
      }
    }
    // afisam totalul
    System.out.println("Total = " + total);
  }
}
```

După rularea exemplului, vom obţine la consolă:

```
1 x Radiera (0.5)
3 x Creion (1.0)
Total = 3.5
```

4.3.5. Compararea obiectelor

Pentru a compara obiecte, putem folosi două interfeţe: `java.lang.Comparable` şi `java.util.Comparator`. Acestea sunt prezentate în continuare.

4.3.5.1. *Interfaţa* `Comparable`

Interfaţa `java.lang.Comparable` este utilizată pentru compararea obiectului curent cu alte obiecte. Se foloseşte un tip generic `T` pentru tipul elementelor cu care se poate compara obiectul curent.

Singura metodă a acestei interfeţe este:

```
int compareTo(T o)
```

Aceasta returnează o valoare negativă, dacă obiectul curent precedă (este mai mic decât) obiectul specificat, zero, dacă obiectele cele două sunt egale, şi o valoarea strict pozitivă, dacă obiectul curent este după (este mai mare decât) obiectul specificat.

Dacă avem `x.compareTo(y) < 0`, atunci trebuie ca `y.compareTo(x) > 0`.

Este recomandat ca, atunci când `x.compareTo(y) == 0`, să avem şi `x.equals(y) == true`.

O clasă nu poate să implementeze de două sau mai multe ori aceeaşi interfaţă, chiar dacă în locul tipului generic sunt folosite tipuri diferite.

Exemplul 4.3.4. Următoarea declaraţie va conduce la o eroare de compilare:

```
public class ClasaDeTest
   implements Comparable<Double>, Comparable<String> {

  public int compareTo(Double d) {
    return 0;
  }

  public int compareTo(String s) {
    return 0;
  }
}
```

Exemplul 4.3.5. În secvenţa de cod de mai jos este definită clasa `Zi`, care are două componente: numărul lunii (`luna`) şi ziua din lună (`zi`). Pentru a putea fi comparate elementele de tip `Zi`, clasa `Zi` implementează interfaţa `Comparable`, cu tipul `Zi`.

```
class Zi implements Comparable<Zi> {
  private int zi, luna;

  public Zi(int zi, int luna) {
    this.zi = zi;
```

```
      this.luna = luna;
    }

    public int compareTo(Zi o) {
      int result = luna - o.luna;
      return result == 0 ? zi - o.zi : result;
    }
}

public class TestComparable {
    public static void main(String[] args) {
      Zi[] zile = new Zi[]{
         new Zi(21, 3),
         new Zi(22, 3),
         new Zi(22, 5)
      };
      System.out.println(zile[0].compareTo(zile[1]));
      System.out.println(zile[1].compareTo(zile[2]));
      System.out.println(zile[0].compareTo(zile[2]));
    }
}
```

În urma execuţiei se va afişa:

```
-1
-2
-2
```

Acest lucru înseamnă că: `Zi(21, 3) < Zi(22, 3) < Zi(22, 5)`.

4.3.5.2. *Interfaţa* `Comparator`

Interfaţa `Comparator` este utilizată pentru compararea a două obiecte. De data aceasta, obiectul curent se ocupă de realizarea comparării. Obiectele trebuie să fie de acelaşi tip. Tipul acestora este utilizat ca tip generic pentru interfaţă şi este notat cu `T`.

Interfaţa `Comparator` conţine doar două metode:

Prototipul metodei	Descrierea metodei
`int` **`compare`**`(T o1, T o2)`	Compară obiectele `o1` şi `o2`. Returnează un număr negativ dacă `o1 < o2`, zero, în caz de egalitate, şi un număr strict pozitiv, dacă `o1 > o2`.
`boolean` **`equals`**`(Object obj)`	Testează egalitatea obiectului curent cu cel transmis ca parametru. Se realizează, de fapt, testarea egalităţii a doi comparatori. Aceştia trebuie să fie egali atunci când impun aceeaşi relaţie de ordine.

Exemplul 4.3.6. Clasa `Produs` este caracterizată de greutate şi preţ. S-au definit doi comparatori: unul care compară produsele după preţ (clasa `ComparaProduseDupaPret`), iar celălalt după greutate (clasa `ComparaProduseDupaGreutate`).

```
import java.util.Comparator;

class Produs {
  int greutate;
  float pret;

  public Produs(int greutate, float pret) {
    this.greutate = greutate;
    this.pret = pret;
  }
}

class ComparaProduseDupaGreutate implements Comparator<Produs> {
  public int compare(Produs p1, Produs p2) {
    return p1.greutate - p2.greutate;
  }
}

class ComparaProduseDupaPret implements Comparator<Produs> {
  public int compare(Produs p1, Produs p2) {
    return (int) Math.signum(p1.pret - p2.pret);
  }
}

public class TestComparator {

  public static void main(String[] args) {
    Produs p1 = new Produs(230, 15.13f);
    Produs p2 = new Produs(330, 10.99f);

    Comparator<Produs> comparator;

    comparator = new ComparaProduseDupaGreutate();
    System.out.println(comparator.compare(p1, p2));

    comparator = new ComparaProduseDupaPret();
    System.out.println(comparator.compare(p1, p2));
  }
}
```

Programul va afişa:

```
-100
1
```

Acest lucru înseamnă că produsul `p1` este înaintea lui `p2`, după criteriul greutăţii, însă `p1` este după `p2`, conform criteriului preţului.

Se observă că definirea metodei `equals()` pentru comparatori este opţională întrucât este moştenită din clasa `Object`.

4.3.6. Mulţimi

Pentru utilizarea mulţimilor de elemente se foloseşte interfaţa `java.util.Set`. Aceasta este derivată din `Collection` şi are în plus restricţia de a nu permite ca elementele din colecţie să se repete.

O clasă care implementează interfaţa `Set`, frecvent utilizată, este `java.util.HashSet`.

Exemplul 4.3.7. Se construieşte o mulţime de numere de tip `Double`, pentru care se calculează suma elementelor.

```
import java.util.HashSet;
import java.util.Iterator;

public class TestHashSet {

  public static void main(String[] args) {
    HashSet<Double> hs = new HashSet<Double>();
    hs.add(3.14159);
    hs.add(-2.57);
    hs.add(1.09);

    // nu mai este adaugat intrucat deja exista
    hs.add(-2.57);

    Iterator<Double> it = hs.iterator();
    double suma = 0.0, element;
    while(it.hasNext()) {
      element = it.next();
      System.out.println(element);
      suma += element;
    }

    System.out.println("Suma este " + suma);
  }
}
```

La execuţie se va afişa:

```
-2.57
3.14159
```

```
1.09
Suma este 1.6615900000000001
```

Se observă că numărul `-2.57` este adăugat o singură dată, deşi metoda de adăugare a fost apelată de două ori pentru acesta.

4.3.7. Liste

Lista reprezintă o colecţie ordonată de elemente. Acestea pot fi accesate direct prin intermediul numărului de ordine (indexul elementului) şi se pot realiza căutări ale elementelor în listă (determinarea indexului). Într-o listă elementele se pot repeta, acestea având indecşi distincţi pentru fiecare apariţie.

Conceptul de listă este dat de interfaţa `java.util.List`. Aceasta foloseşte pentru elementele listei un tip generic notat cu `E`.

Interfaţa `List` este derivată din interfaţa `Collection`. În plus, interfaţa `List` conţine următoarele metode:

Prototipul metodei	Descrierea metodei
`void add(int index, E element)`	Inserează elementul `element` pe poziţia `index`.
`E get(int index)`	Returnează elementul cu indexul specificat.
`int indexOf(Object o)`	Returnează poziţia primei apariţii a elementului specificat în listă. Dacă elementul nu există în listă se va obţine valoarea `-1`.
`int lastIndexOf(Object o)`	Obţine poziţia ultimei apariţii a obiectului o în listă sau `-1`, dacă obiectul nu face parte din listă.
`ListIterator<E> listIterator()`	Obţine un iterator pentru lista curentă.
`ListIterator<E> listIterator(int index)`	Obţine un iterator pentru lista curentă care o va parcurge începând cu poziţia `index`.
`E remove(Object o)`	Elimină obiectul din listă. Se returnează obiectul eliminat.
`E set(int index, E element)`	Înlocuieşte elementul de la poziţia `index` cu elementul `element`. Se returnează vechiul element de la poziţia specificată.
`List<E> subList(int fromIndex, int toIndex)`	Extrage sublista cuprinsă între poziţiile `fromIndex` şi `toIndex`.

Metodele care primesc un index invalid aruncă excepţia `IndexOutOfBoundsException`.

Metoda `listIterator()` returnează un iterator pentru liste de elemente, dat de interfaţa `ListIterator`. Aceasta este derivată din `Iterator` şi posedă următoarele metode:

Prototipul metodei	Descrierea metodei
`void` **`add`**`(E e)`	Inserează elementul specificat înaintea elementului următor.
`boolean` **`hasNext`**`()`	Returnează `true` dacă mai urmează elemente.
`boolean` **`hasPrevious`**`()`	Returnează `true` dacă sunt elemente înaintea celui care urmează.
`E` **`next`**`()`	Returnează elementul următor.
`int` **`nextIndex`**`()`	Returnează indexul elementului următor.
`E` **`previous`**`()`	Obţine elementul precedent.
`int` **`previousIndex`**`()`	Returnează indexul elementului precedent.
`void` **`remove`**`()`	Elimină elementul returnat ultima dată de metoda `next()` sau `previous()`.
`void` **`set`**`(E e)`	Înlocuieşte elementul returnat ultima dată de metoda `next()` sau `previous()` cu elementul specificat.

Există două clase utilizate frecvent care implementează interfaţa `List`. Acestea sunt `java.util.ArrayList` şi `java.util.Vector`. Diferenţa majoră dintre cele două clase constă în faptul că `Vector` este sincronizată (poate fi folosită fără probleme de mai multe fire de execuţie), iar `ArrayList` nu este sincronizată.

Exemplul 4.3.8. Se creează un vector cu informaţii despre studenţi.

```
import java.util.Vector;

class Student {
   int numarMatricol;
   String nume, prenume;
   float media;

  Student(int numarMatricol, String nume, String prenume, float
   media) {
     this.numarMatricol = numarMatricol;
     this.nume = new String(nume);
     this.prenume = new String(prenume);
     this.media = media;
   }

   public String toString() {
     return("NumarMatricol:" + numarMatricol +
            "\nNume: " + nume +
            "\nPrenume: " + prenume +
            "\nMedia: " + media);
```

```
    }
} // sfarsitul clasei Student

public class VectorStudenti {
  public static void main(String args[]) {
    Vector<Student> listaStudenti = new Vector<Student>();
    listaStudenti.addElement(new Student(52321, "Trufanda",
      "Viorel", 9.99f));
    listaStudenti.addElement(new Student(52322, "Raicu",
      "Florin", 8.99f));
    listaStudenti.addElement(new Student(52426, "Munteanu",
      "Loredana", 9.49f));
    for (int i = 0; i < listaStudenti.size(); i++) {
       Student stud = listaStudenti.get(i);
       System.out.print("\nStudent " + i + "\n");
       System.out.println(stud);
    }
  }
} // sfarsitul clasei VectorStudenti
```

Exemplul 4.3.9. Conversia dintr-un şir de întregi (`int`) la un `Vector` de `Integer`:

```
import java.util.Vector;

public class UtilizareVectori {
  public static void main(String args[]) {
    int tablouFix [] = {1, 3, 0, 5, 3, 0, 3, 4, 0, 2, 0, 0,
      5, 12345};
    Vector<Integer> listaIntregi = new Vector<Integer>();
    // varibila listaIntregi va contine elementele din
    // tablouFix[] care sunt nenule
    // se parcurge sirul de intregi
    for (int i = 0; i < tablouFix.length; i++) {
      if (tablouFix[i] != 0)
        // daca elementul este nenul
        // se adauga in lista de intregi
        listaIntregi.add(tablouFix[i]);
    }
    // afisarea listei de intregi
    for (int i = 0; i < listaIntregi.size(); i++) {
      // se obtine elementul de pe pozitia i
      int valoare = listaIntregi.get(i);
      System.out.println("Element " + i + ": " + valoare);
    }
  }
} // sfarsitul clasei UtilizareVectori
```

Exemplul 4.3.10. Evidenţierea modului de utilizare a vectorilor de obiecte de tipuri diferite:

```
import java.util.Vector;

class ClasaGenerica {
  String nume;

  ClasaGenerica(String nume) {
    this.nume = nume;
  }

  public String toString() {
    return(nume);
  }
} // sfarsitul clasei ClasaGenerica

public class VectorGeneric {
  public static void main(String args[]) {
    Vector<Object> listaGenerica = new Vector<Object>();

    // popularea vectorului
    listaGenerica.add(new ClasaGenerica("Laguna"));
    listaGenerica.add(112);
    listaGenerica.add("abc");

    // parcurgerea vectorului
    for (int i = 0; i < listaGenerica.size(); i++) {
      Object obiectCurent = listaGenerica.get(i);
      System.out.println("Obiect din clasa " +
          obiectCurent.getClass().getName() +
          ": " + obiectCurent);
      if (obiectCurent instanceof Integer) {
        Integer nr = (Integer) obiectCurent;
        System.out.println("\tAm gasit numarul " + nr);
      }
    }
  }
} // sfarsitul clasei VectorGeneric
```

Execuţia programului va avea ca efect:

```
Obiect din clasa ClasaGenerica: Laguna
Obiect din clasa java.lang.Integer: 112
```

```
        Am gasit numarul 112
Obiect din clasa java.lang.String: abc
```

Clasa `java.util.Stack` este derivată din clasa `Vector` şi reprezintă conceptul de stivă (LIFO – *Last Input First Output*).

Clasa `Stack` are un singur constructor fără parametri şi defineşte încă cinci metode:

Prototipul metodei	Descrierea metodei
`boolean` **`empty`**`()`	Testează dacă stiva este vidă.
`E` **`peek`**`()`	Întoarce obiectul din vârful stivei fără să-l elimine.
`E` **`pop`**`()`	Întoarce obiectul din vârful stivei şi-l şterge din stivă.
`E` **`push`**`(E item)`	Inserează un obiect în stivă.
`int` **`search`**`(Object o)`	Returnează prima poziţie în care este găsit obiectul în stivă. Întregul returnat reprezintă poziţia relativă faţă de vârful stivei sau `-1`, dacă elementul nu este găsit.

Exemplul 4.3.11. O variantă de utilizare a unei stive cu obiecte de tipuri diferite:

```
import java.util.ListIterator;
import java.util.Stack;

public class TestStiva {
  public static void main(String args[]) {
    Stack<Object> stiva = new Stack<Object>();

    stiva.push(new String("unu"));
    stiva.push(new Integer(2));
    stiva.push(3.0);

    System.out.println("elementul din top este " +
         stiva.peek());

    ListIterator<Object> it = stiva.listIterator();
    while (it.hasNext()) {
      System.out.println(" " + it.next());
    }
  }
} // sfarsitul clasei testStiva
```

Se va afişa în urma excuţiei:

```
elementul din top este 3.0
 unu
 2
 3.0
```

4.3.8. Clasa `BitSet`

Clasa `BitSet` este derivată direct din `Object` pentru a implementa structura de mulţime folosind un vector caracteristic (`boolean`). Astfel, fiecare componentă are o valoare booleană (`true`, `false`). Biţii pot fi examinaţi, setaţi sau şterşi. Biţii dintr-un `BitSet` sunt indexaţi după întregi pozitivi. Un `BitSet` poate fi utilizat pentru a schimba conţinutul unui alt `BitSet` folosind operatorii logici AND, OR inclusiv şi OR exclusiv (XOR). Implicit, toţi biţii dintr-un set au valoarea `false`. Fiecare `BitSet` are o dimensiune curentă, care coincide cu numărul de biţi din `BitSet`.

Există doi constructori:

- `public BitSet();`
- `public BitSet(int numarBiti);`

A doua formă permite specificarea dimensiunii iniţiale a obiectului `BitSet`.

Clasa `BitSet` are următoarele metode:

Prototipul metodei	Descrierea metodei
`void` **`and`**`(BitSet set)`	Realizează AND logic între mulţimea curentă şi cea dată ca parametru. Rezultatul este stocat în mulţimea curentă.
`void` **`andNot`**`(BitSet set)`	Şterge biţii setaţi în mulţimea curentă, care se regăsesc în mulţimea dată ca parametru.
`int` **`cardinality`**`()`	Returnează numărul de biţi setaţi în mulţime.
`void` **`clear`**`()`	Şterge toate elementele setate din mulţime.
`void` **`clear`**`(int index)`	Setează bitul specificat la `false`.
`void` **`clear`**`(int fromIndex, int toIndex)`	Setează cu `false` biţi de la `fromIndex` (inclusiv) până la `toIndex` (exclusiv).
`Object` **`clone`**`()`	Realizează o copie a obiectului curent.
`boolean` **`equals`**`(Object obj)`	Verifică dacă obiectul curent este egal cu cel dat ca parametru.
`void` **`flip`**`(int bitIndex)`	Modifică bitul de pe poziţia `bitIndex` cu complementul său.
`void` **`flip`**`(int fromIndex, int toIndex)`	Modifică biţii de pe poziţiile cuprinse între `fromIndex` şi `toIndex` (exclusiv) cu complementele lor.
`boolean` **`get`**`(int bitIndex)`	Obţine valoarea bitului cu indexul `bitIndex`.
`BitSet` **`get`**`(int fromIndex, int toIndex)`	Extrage submulţimea cuprinsă între `fromIndex` şi `toIndex` (exclusiv).

→

➔

`int` **`hashCode`**`()`	Returnează codul hash pentru obiectul curent.
`boolean` **`intersects`**`(BitSet set)`	Returnează `true` dacă mulţimea curentă şi cea dată ca parametru au măcar un bit setat pe aceeaşi poziţie.
`boolean` **`isEmpty`**`()`	Returnează `true` dacă toţi biţii au valoarea `false`.
`int` **`length`**`()`	Returnează cea mai mare poziţie pentru care bitul este `true` plus o unitate.
`int` **`nextClearBit`**`(int fromIndex)`	Returnează poziţia primului bit aflat după poziţia `fromIndex` şi care are valoarea `false`.
`int` **`nextSetBit`**`()`	Returnează poziţia primului bit aflat după poziţia `fromIndex` şi care are valoarea `true`.
`void` **`or`**`(BitSet set)`	Efectuează operaţia OR logic între mulţimea curentă şi cea specificată.
`void` **`set`**`(int bitIndex)`	Bitul de pe poziţia `bitIndex` este iniţializat cu valoarea `true`.
`void` **`set`**`(int bitIndex, boolean value)`	Bitul de pe poziţia `bitIndex` este iniţializat cu valoarea dată de `value`.
`void` **`set`**`(int fromIndex, int toIndex)`	Iniţializează cu valoarea `true` biţii de pe poziţiile cuprinse între poziţiile `fromIndex` inclusiv şi `toIndex` exclusiv.
`void` **`set`**`(int fromIndex, int toIndex, boolean value)`	Iniţializează cu valoarea dată de `value` biţii de pe poziţiile cuprinse între poziţiile `fromIndex` inclusiv şi `toIndex` exclusiv.
`int` **`size`**`()`	Returnează numărul de biţi din mulţime.
`String` **`toString`**`()`	Obţine un şir de caractere care reprezintă mulţimea indecşilor care au biţii cu valoarea `true`.
`void` **`xor`**`(BitSet set)`	Efectuează operaţia XOR logic (sau exclusiv) între mulţimea curentă şi cea specificată.

Exemplul 4.3.12. Programul ilustrează o modalitate de utilizare a clasei `BitSet`:

```
import java.util.BitSet;

public class BitSetTest {
  public static void main(String args[]) {
      BitSet bs1 = new BitSet(4);
      BitSet bs2 = new BitSet(4);

      // setam bs1 = (true, false, true, true)
      bs1.set(0);
      bs1.clear(1);
      bs1.set(2, 4);
      System.out.println(bs1); // se va afisa {0, 2, 3}

      // setam bs1 cu toti bitii activati
      for (int i = 0; i < 4; i++)
        bs2.set(i);

      // inverseaza bs1 cu bs2, folosind metoda XOR
      bs1.xor(bs2);
      System.out.println(bs1); // se va afisa {1}
    }
  } // sfarsitul clasei BitSetTest
```

4.3.9. Clasa `Properties`

Clasa `Properties` este subclasă a clasei `Hashtable` şi este utilizată de sistemul de execuţie a programelor Java pentru lucrul cu un tip special de tabel hash, aşa-numita listă de proprietăţi. Clasa `Properties` lucrează cu o colecţie de perechi de şiruri cheie/valoare.

Listarea proprietăţilor sistemului se poate realiza prin apelul funcţiei `getProperties()` din clasa `System`:

```
public static Properties getProperties();
```

Acestea reprezintă valori care descriu mediul de execuţie curent (numele utilizatorului, numele sistemului de operare, directorul rădăcină etc.).

Exemplul 4.3.13. Programul de mai jos va lista toate proprietăţile:

```
public class Proprietati {
  public static void main(String args[]) {
    System.getProperties().list(System.out);
  }
}
```

Atunci când programele operează cu obiecte `Hashtable`, va trebui uneori să stocăm şi să preluăm conţinutul unei tabele dintr-un fişier. O listă de proprietăţi permite stocarea de valori implicite. Dacă un program nu poate găsi o anumită proprietate într-o listă, lista poate încerca valoarea implicită din fişierul de proprietăţi.

Limitarea unei liste de proprietăţi este că atât cheia, cât şi valoarea stocată în tabelă trebuie să fie obiecte `String`.

Aplicaţiile posibile ale listei de proprietăţi sunt cele de personalizare şi salvare a configuraţiei între execuţiile unui program. Fişierul listă de proprietăţi este „echivalent" cu un fişier `.ini` utilizat sub Windows. Sintaxa este asemănătoare.

Exemplul 4.3.14. Iată un fişier listă de proprietăţi denumit `propApl.ini`:

```
#lista de proprietati  propApl.ini
Foreground=0000FF
Background=FF8888
FontSize=12
Font=Arial
```

Crearea unei liste de proprietăţi se face apelând unul dintre constructorii clasei `Properties`:

- `public Properties()` – creează o listă de proprietăţi vidă fără valori implicite;
- `public Properties(Properties defaults)` – creează o listă de proprietăţi vidă cu valorile iniţiale date de `defaults`.

În tabelul de mai jos prezentăm câteva metode utile din clasa `Properties`.

Prototipul metodei	Descrierea metodei
`Object` **`setProperty`**`(String key, String value)`	Apelează metoda `put()` din superclasa `Hashtable` pentru adăugarea perechii (`key`, `value`).
`String` **`getProperty`**`(String key)`	Returnează valoarea asociată cheii `key` sau `null`, dacă cheia nu este găsită.
`String` **`getProperty`**`(String key, String defaultValue)`	Returnează valoarea asociată cheii `key` sau `defaultValue`, în cazul în care cheia nu este găsită.
`void` **`list`**`(PrintStream out)`	Scrie proprietăţile în fluxul de octeţi specificat.
`void` **`list`**`(PrintWriter out)`	Scrie proprietăţile în fluxul de caractere specificat.
`void` **`load`**`(InputStream inStream)`	Citeşte proprietăţile din fluxul de octeţi `inStream`.
`void` **`load`**`(Reader reader)`	Citeşte proprietăţile din fluxul de caractere `reader`.
`void` **`loadFromXml`**`(InputStream in)`	Încarcă proprietăţile date în format XML din fluxul de octeţi `in`.
`void` **`store`**`(OutputStream out, String comments)`	Trimite proprietăţile în fluxul de octeţi `out`. Şirul de caractere `comments` conţine un comentariu care va fi salvat sau `null`, dacă nu se doreşte specificarea comentariului.
`void` **`store`**`(Writer writer, String comments)`	Trimite proprietăţile în fluxul de caractere `writer`.

➔

➔

void **storeToXml**(OutputStream os, String comment)	Trimite proprietăţile în format XML către fluxul de octeţi `os`.
void **storeToXml**(OutputStream os, String comment, String encoding)	Trimite proprietăţile în format XML către fluxul de octeţi `os`. `String`-ul `encoding` stabileşte setul de caractere utilizat în formatul XML.
Set<String> **stringPropertyNames**()	Returnează mulţimea cheilor din tabelul asociativ.

Formatul XML utilizat pentru proprietăţi este conform DTD-ului de la adresa *http://java.sun.com/dtd/properties.dtd*.

Exemplul 4.3.15. Programul reprezintă un model de utilizare a listelor de proprietăţi:

```
import java.util.Iterator;
import java.util.Properties;

public class ListaProprietati {
  public static void main(String args[]) {
    Properties prop = new Properties();

    // adaugam cateva asocieri
    prop.setProperty("unu", "1");
    prop.setProperty("doi", "2");
    prop.setProperty("trei", "3");
    prop.setProperty("patru", "4");
    prop.setProperty("cinci", "5");

    // parcurgem multimea de chei
    Iterator<String> it = prop.stringPropertyNames().iterator();
    while (it.hasNext()) {
      String cheieProp = it.next();
      System.out.println(cheieProp + " = " +
        prop.getProperty(cheieProp));
    }
  }
}
```

Rularea acestui program va avea drept consecinţă afişarea pe ecran:

```
doi = 2
patru = 4
trei = 3
unu = 1
cinci = 5
```

Nu este garantat că ordinea listei enumerate va fi aceeaşi cu cea de adăugare a elementelor în lista de proprietăţi (cum se întâmplă în cazul clasei `Hashtable`).

Exemplul 4.3.16. Salvarea proprietăţilor într-un fişier:

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Properties;

public class SalvareProprietati {

  public static void main(String[] args) {
    Properties config = new Properties();

    config.setProperty("port", "9090");
    config.setProperty("maxUsers", "100");
    config.setProperty("maxMsgSize", "2048");

    FileOutputStream fluxOut = null;

    try {
      fluxOut = new FileOutputStream("config.ini");
      config.store(fluxOut, "Fisier de configurare");
      fluxOut.close();
    } catch (IOException e) {
      System.err.println(e.getMessage());
    }
  }
}
```

În urma execuţiei programului de mai sus, conţinutul fişierului `config.ini` va fi:

```
#Fisier de configurare
#Thu Oct 12 22:06:07 EEST 2006
maxUsers=100
port=9090
maxMsgSize=2048
```

Exemplul 4.3.17. Următoarea aplicaţie citeşte proprietăţile generate în exemplul precedent:

```
import java.io.FileInputStream;
import java.io.IOException;
import java.util.Properties;

public class IncarcareProprietati {

  public static void main(String[] args) {
    Properties props = new Properties();
    // setam cateva proprietati
    props.setProperty("bufSize", "512");
    props.setProperty("maxUsers", "20");

    // incarcam proprietatile din fisier
    try {
      FileInputStream fluxIn = new FileInputStream("config.ini");
      props.load(fluxIn);
      fluxIn.close();
    } catch (IOException e) {
      e.printStackTrace();
    }

    // afisam proprietatile din tabelul de proprietati
    props.list(System.out);
  }
}
```

Execuţia programului va avea ca efect afişarea la consolă a proprietăţilor:

```
-- listing properties --
maxUsers=100
port=9090
bufSize=512
maxMsgSize=2048
```

Se observă că proprietatea `bufSize` a fost păstrată, iar `maxUsers` a fost suprascrisă.

Exemplul 4.3.18. Aplicaţia următoare salvează proprietăţile într-un fişier XML. (Vezi şi capitolul 12.)

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Properties;

public class SalvareProprietatiXML {

  public static void main(String[] args) {
    Properties config = new Properties();

    config.setProperty("timeout", "5000");
    config.setProperty("heIloMsg", "Hello!");

    FileOutputStream fluxOut = null;

    try {
      fluxOut = new FileOutputStream("config.xml");
```

```
         config.storeToXML(fluxOut, "Fisier de configurare");
         fluxOut.close();
      } catch (IOException e) {
         System.err.println(e.getMessage());
      }
   }
}
```

Conţinutul fişierului `config.xml` va fi:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/
   properties.dtd">
<properties>
<comment>Fisier de configurare</comment>
<entry key="timeout">5000</entry>
<entry key="helloMsg">Hello!</entry>
</properties>
```

4.4. Concluzii

Capitolul de faţă a debutat cu prezentarea a trei clase de bază pentru crearea, afişarea şi manipularea şirurilor de caractere: `String`, `StringBuffer`, `StringBuilder` şi `StringTokenizer`. Se observă faptul că operaţiile cu şiruri de caractere se pot realiza foarte uşor utilizând clasele prezentate.

Structurile dinamice sunt frecvent utilizate în practică pentru uşurinţa şi eficienţa acestora. S-a prezentat lucrul cu enumeratori, colecţii, liste, tabele de asocieri, stive, mulţimi şi proprietăţi. Clasele şi interfeţele corespunzătoare structurilor menţionate contribuie la accelerarea procesului de dezvoltare a aplicaţiilor Java.

4.5. Test-grilă

Întrebarea 4.5.1. Un obiect de tip `String` permite parcurgerea caracter cu caracter a literalului stocat?

a) Există o metodă care se ocupă de lucrul acesta.
b) Se poate realiza cu ajutorul mai multor metode din clasa `String`.
c) Se poate realiza doar prin intermediul clasei `StringBuffer`.
d) Această operaţie poate fi efectuată şi cu ajutorul unui obiect de tip `StringTokenizer`.

Întrebarea 4.5.2. Ce se poate spune despre următoarea secvenţă de cod?

```
String s = "String";
```

```
s.toString().toString.toString();
```

a) Se obţine eroare la compilare, deoarece nu s-a utilizat operatorul `new` pentru obiectul `s`.
b) Apare o eroare la compilare, întrucât obiectul `s` va conţine numele unui cuvânt rezervat.
c) Se va genera o eroare la execuţie, pentru că metoda `toString()` returnează o valoare care nu este preluată.
d) Codul este corect din punct de vedere sintactic.
e) Se apelează metoda `toString()` pentru acelaşi obiect.

Întrebarea 4.5.3. Care dintre metodele clasei `String` de mai jos returnează un obiect de tip `String`?

a) `replace()`
b) `split()`
c) `intern()`
d) `toLowerCase()`

Întrebarea 4.5.4. Ce va afişa la consolă următorul apel?

```
System.out.println(10 + 9 + new StringBuffer("opt") + 7 + 6 + "5")
```

a) `109opt765`
b) `19opt135`
c) `19opt18`
d) `19opt765`
e) Se obţine o eroare la compilare sau la execuţie.

Întrebarea 4.5.5. Pentru a gestiona numele unor persoane, care structură este mai eficientă pentru căutare?

a) o colecţie de tip vector;
b) o tabelă *hash*;
c) o listă dublu înlănţuită;
d) un arbore binar;
e) o mulţime.

4.6. Exerciţii propuse spre implementare

Exerciţiul 4.6.1. Utilizând clasa `Stack`, scrieţi un program Java care simulează un calculator de buzunar. Implementaţi operaţiile + (unar şi binar), *, / şi radical. Testaţi condiţia de pozitivitate la radical (returnaţi un mesaj propriu de eroare), cazul împărţirii la `0` sau când se obţine un număr care nu este în domeniu.

Exerciţiul 4.6.2. Scrieţi o clasă proprie care extinde clasa `Stack` şi care implementează afişarea, numărarea, oglindirea elementelor unei stive.

Exerciţiul 4.6.3. Folosind clasa `BitSet`, scrieţi un program Java care execută reuniunea, intersecţia a două mulţimi, precum şi complementara a două mulţimi. Mulţimile se citesc din două fişiere şi au elemente din mulţimea `{0, ..., k}`, `k` fixat, strict pozitiv (de exemplu, `k = 1000`).

5. Fişiere, fluxuri de date şi serializarea obiectelor

5.1. Cuvinte-cheie

- fişiere, fluxuri de intrare/ieşire, buffere, filtre
- serializare, deserializare
- procesarea fişierelor

5.2. Introducere

Fişierele de intrare şi ieşire Java implică utilizarea fluxurilor (engl. *streams*). Un *flux* este o reprezentare abstractă a unui dispozitiv de intrare sau ieşire care este sursă, respectiv destinaţie pentru date. Astfel, putem scrie date într-un flux sau citi date dintr-un flux. De fapt, putem vedea fluxul ca o secvenţă de octeţi care „curge" (engl. *flows*) într-un program Java.

Când se scriu date într-un flux, fluxul se numeşte *flux de ieşire*. Fluxul de ieşire poate circula către orice dispozitiv unde se poate transfera o secvenţă de octeţi, cum ar fi un fişier de pe un hard disk local, un fişier de pe un sistem de la distanţă sau o imprimantă. Se poate direcţiona un flux de ieşire către ecran, dar acesta va afişa doar informaţii textuale, nu şi grafice, care necesită un suport specializat (vom reveni în capitolul de ferestre grafice).

Putem citi date dintr-un *flux de intrare*. Acesta poate fi orice sursă serială de date (de obicei, un fişier pe hard diskul local, tastatură) sau de pe un calculator de la distanţă.

Fără a intra în detalii de sisteme de operare, în principiu există două posibilităţi de memorare a datelor (informaţiilor) în calculatorul local:

a) *memoria internă* (engl. RAM - *Random Access Memory*);
b) *memoria externă*, cum ar fi dispozitivele de memorare a fişierelor (USB stick, hard disk, CD, DVD etc.).

De obicei, fluxurile de intrare şi ieşire ale maşinii pe care se execută aplicaţia sunt disponibile în aplicaţii Java, nu şi appleturilor. În caz contrar, un applet apelat dintr-o pagină Web ar putea accesa hard diskul local creând eventuale probleme de securitate. Încercarea de a accesa fişiere ale discului local dintr-un applet Java este prohibită, de obicei, prin aruncarea unei excepţii `IOException`. Cu toate acestea, directorul în care se află appletul şi subdirectoarele acestuia sunt accesibile pentru applet. În plus, se pot folosi proprietăţile de securitate Java pentru controlul unui applet (şi a unei aplicaţii care rulează sub un manager de securitate) putând accesa fişiere şi alte resurse pentru care există permisiune explicită.

Principalul motiv al utilizării unui flux ca bază pentru operaţiile de intrare/ieşire în Java este independenţa codului faţă de dispozitivul implicat. Avantajele ar fi *transparenţa*, nu trebuie să ştim detaliile fizice ale fiecărui dispozitiv, şi *portabilitatea*, programul va funcţiona pe diferite dispozitive fără a schimba sursa sau codul obiect Java.

Metodele fluxurilor de intrare/ieşire permit citirea/scrierea unei cantităţi mici de date (un octet, un caracter etc.), deci transferul datelor în acest mod către un dispozitiv fizic (de exemplu, un hard disk) ar fi extrem de ineficient. Pentru rezolvarea acestei probleme, un flux are ataşat un buffer (engl. *buffered stream*). Un *buffer* este un bloc de memorie folosit pentru a reţine date din/către un dispozitiv extern.

Fluxurile cu buffer asigură că transferul datelor între memorie şi un dispozitiv extern folosesc un buffer suficient de mare pentru ca operaţiile de intrare/ieşire să fie eficiente. Când se scrie într-un flux de ieşire bufferizat, datele sunt trimise către buffer şi nu direct către dispozitivul extern. Cantitatea de date din buffer este automat transferată şi când acesta s-a umplut, datele sunt transferate către dispozitivul extern. Uneori se doreşte ca datele să fie trimise către dispozitiv înaintea umplerii complete a bufferului. Există metode care eliberează conţinutul bufferului (înainte ca acesta să fie plin) şi trimit datele către dispozitivul extern. Această operaţie se numeşte golirea bufferului (engl. *flushing the buffer*). Fluxurile de intrare bufferizate lucrează la fel, adică orice operaţie de citire dintr-un flux de intrare bufferizat va citi date din buffer, iar când acesta va fi gol, se vor transfera date din dispozitivul extern către buffer (dacă sunt suficiente date).

Pachetul `java.io` oferă două tipuri de fluxuri: *binare* (care conţin date binare) şi *caracter* (care conţin date caracter). În cazul fluxurilor binare, scrierea/citirea datelor într-un flux se face ca o serie de octeţi, exact cum apar acestea în memorie. Nu este nevoie de vreo transformare a datelor. Java utilizează scrierea internă a caracterelor Unicode, aşadar când se scriu caractere către un flux ca date binare, fiecare caracter pe 16 biţi este reprezentat pe doi octeţi, cel semnificativ fiind scris primul.

Fluxurile caracter sunt folosite pentru memorarea şi obţinerea textului sau pentru a citi un fişier text care nu a fost scris de un program Java. Toate datele numerice sunt convertite în reprezentări text înainte de a fi scrise în flux. Aceasta implică formatarea datelor pentru a genera reprezentarea caracter a valorilor datelor. Citirea datelor numerice dintr-un flux care conţine text implică mai multă muncă decât citirea datelor binare. Citirea dintr-un flux presupune că se cunoaşte câte caractere formează o valoare numerică şi converteşte corect caracterele din flux. Trebuie să se recunoască începutul şi sfârşitul fiecărei valori numerice şi apoi se realizează conversia atomului lexical (engl. *token*), adică a secvenţei de caractere care reprezintă acea valoare, în formă binară a valorii.

Exemplul 5.2.1. Presupunem că avem declaraţiile:

```
int i = 123, j = 45;
```

Valorile acestor variabile sunt reprezentate în memorie (pe opt octeţi) ca: `0x7B`, respectiv `0x2D`. În momentul trimiterii către un flux se va converti fiecare cifră a valorii întregi în caracter ASCII, obţinându-se secvenţa de cinci octeţi: `0x31`, `0x32`, `0x33`, respectiv `0x34`, `0x35`. Pentru conversia inversă va trebui să ştim că primii trei octeţi se referă la prima valoare întreagă (formată din cifrele `1`, `2`, `3`) şi următorii doi octeţi la cea de-a doua valoare (formată din cifrele `4`, `5`).

Când se scriu caractere într-un flux ca date caracter, caracterele Unicode sunt convertite automat la reprezentarea locală a caracterelor (folosite de maşina gazdă) şi apoi acestea sunt scrise în fişier. Când se citeşte un şir de caractere, datele din fişier sunt convertite automat din reprezentarea maşinii locale în caractere Unicode. În cazul fluxurilor caracter, programul citeşte şi scrie caractere Unicode, dar fişierul va conţine caractere din codificarea caracterelor echivalentă utilizată de calculatorul local. Fluxul suport este întotdeauna o secvenţă de octeţi, indiferent dacă se scriu/citesc date binare sau caractere.

Java utilizează două feluri de reprezentări text şi codificare a caracterelor:

- Unicode – pentru reprezentarea internă a caracterelor şi a `String`-urilor;
- UTF – pentru intrări şi ieşiri.

Unicode utilizează 16 biţi pentru reprezentarea unui caracter. În cazul în care cei mai semnificativi nouă biţi sunt toţi 0, atunci codificarea este simplă – standard ASCII (ultimii 7 biţi fiind reprezentarea caracterelor). Altfel, biţii reprezintă un caracter care nu este reprezentat în ASCII pe 7 biţi. Tipul `char` din Java foloseşte codificarea Unicode (de asemenea, clasa `String`).

Codificarea Unicode este suficientă pentru majoritatea limbilor, dar cele asiatice grafice reprezintă o problemă (sunt prea multe caractere grafice). Soluţia constă în utilizarea UTF (*Universal Character Set Transformation Format*). Codificarea UTF foloseşte exact numărul dorit de biţi: mai puţini pentru alfabete mici, mai mulţi pentru alfabete mari.

O codificare a caracterelor este o corespondenţă bijectivă dintre mulţimea caracterelor şi o mulţime de numere binare. Fiecare platformă Java are o codificare a caracterelor implicită, care este folosită pentru interpretarea dintre Unicode intern şi octeţi externi. Codificarea implicită a caracterelor reflectă limba şi cultura locală. (Vezi capitolul 15.)

Exemplul 5.2.2. Următorul subprogram Java va returna numele codificării caracterelor de pe platforma Java (de exemplu `Cp1250`).

```
try {
  FileInputStream fis = new FileInputStream("fisier.in");
  InputStreamReader isr = new InputStreamReader(fis);
  System.out.println("Codificarea este " + isr.getEncoding());
}
catch (IOException e) {
  System.err.println("Eroare scriere fisier " + e);
}
```

Când are loc o operaţie de intrare/ieşire în Java, sistemul trebuie să ştie care este codificarea caracterelor. Clasele I/O utilizează, de obicei, codificarea implicită locală, în caz că nu se precizează una explicită. Cu toate că este adecvată codificarea implicită locală, în aplicaţiile de reţea, când se comunică cu un alt computer, acestea trebuie să folosească aceeaşi codificare. În acest caz, este recomandată cererea explicită 8859-1.

5.3. Clase Java pentru intrări şi ieşiri

Toate clasele de intrare/ieşire se găsesc în pachetul `java.io`. Vom începe prin prezentarea claselor `File` şi `RandomAccessFile` care furnizează funcţionalitate pentru navigare pe sistemul local, descriind fişierele, directoarele şi accesul direct (nesecvenţial) la fişiere. Accesul secvenţial al fişierelor se realizează cu *fluxuri de citire* (engl. *readers*) şi *fluxuri de scriere* (engl. *writers*) pe care le vom prezenta ulterior.

5.3.1. Clasa `File`

Clasa `File` pune la dispoziţie facilităţi pentru manipularea fişierelor şi directoarelor. Programele Java care folosesc această clasă trebuie să conţină instrucţiunea de import:

```
import java.io.File;
```

sau pentru importarea claselor din întreg pachetul:

```
import java.io.*;
```

Clasa `java.io.File` reprezintă numele unui fişier sau director care poate exista pe sistemul de fişiere a maşinii gazdă. O cale în structura de directoare (sub Windows) se afişează folosind separatorul „\" (engl. *backslash*), cum ar fi, de exemplu, `c:\temp\dil.xml`. Pe un sistem de operare UNIX, calea se scrie cu „/" (engl. *slash*), cum ar fi `/home/stefan/java/TestGrila.java`.

Acest separator se poate determina folosind constanta (din clasa `File`):

```
public final static String separator;
```

Calea este absolută în exemplele precedente, deoarece este specificată de la directorul rădăcină până la fişier. Dacă, de exemplu, suntem în directorul `stefan`, atunci calea relativă este `java/testGrila.java`. Cea mai simplă formă de constructor pentru aceasta clasă este:

```
File(String numeCale);
```

Construcţia unei instanţe a lui `File` nu creează un fişier pe sistemul local, ci doar o instanţă care încapsulează şirul specificat. De exemplu, iată cum putem declara şi crea o instanţă a clasei `File`:

```
File fis = new File("c:\\temp\\dil.xml");
```

Reamintim că \\ înseamnă \, deoarece \ este un caracter cu semnificaţie specială (engl. *escape*) şi separator de fişiere în sistemul de operare Windows.

Mai există două variante de constructori din clasa `File`:

a) `File(String dir, String subCale);`

De exemplu, furnizăm un director şi o cale absolută, astfel:

```
File f = new File("/tmp", "xyz");
```

b) `File(File dir, String subCale);` de exemplu, furnizăm o instanţă a lui `File` şi o cale relativă, astfel:

```
File f1 = new File("C:\\tmp");
File f2 = new File(f1, "xyz.java");
```

Iată câteva metode utile pentru lucrul cu fişiere:

- `public String getPath();`

Furnizează calea relativă; exemplu:

```
String caleRelativa = fis.getPath();
```

- `public String getAbsolutePath();`
  Furnizează calea absolută; exemplu:

```
String caleAbsoluta = fis.getAbsolutePath();
```

- `public String getCanonicalPath();`
  Furnizează calea canonică a unui fişier sau director. Este similară cu metoda `getAbsolutePath()`, dar sunt rezolvate (identificate) simbolurile `.` (directorul curent) şi `..` (directorul părinte).
- `public boolean exists();`
  Returnează `true`, dacă fişierul sau directorul specificat există.
- `public boolean isDirectory();`
  Întoarce `true`, dacă fişierul există şi este director.
- `public boolean isFile();`
  Returnează `true`, dacă fişierul există pe sistemul de fişiere şi este un fişier ordinar.
- `public long length();`
  Returnează lungimea fişierului în octeţi sau `0`, dacă nu există; exemplu:

```
long lungime = fis.length();
```

- `public String[] list();`
  Returnează un şir de nume de fişiere din directorul specificat de obiectul `File`; exemplu:

```
String [] continutDirector = fis.list();
```

Dacă fişierul `fis` nu este director, atunci funcţia va întoarce `null`;

- `boolean canRead();`
  Returnează `true`, dacă fişierul sau directorul poate fi citit.
- `boolean canWrite();`
  Returnează `true`, dacă fişierul sau directorul poate fi modificat.
- `boolean delete();`
  Returnează `true`, dacă fişierul sau directorul a fost şters.
- `boolean mkdir();`
  Returnează `true`, dacă s-a reuşit crearea unui director a cărui cale este descrisă de obiectul `File`.
- `boolean renameTo(File noulNume);`
  Returnează `true`, dacă s-a putut redenumi fişierul sau directorul, altfel returnează `false`.

Exemplul 5.3.1. Următorul program pune în evidenţă crearea (în caz că nu există deja) a directorului `C:\dirProba` (sub sistemul Windows):

```
import java.io.File;

public class Fisiere {
  public static void main(String args[]) {
      boolean reusita;
      File dirNou = new File("C:\\dirProba");
      if (dirNou.isDirectory())
        System.out.println(
          "Directorul C:\\dirProba exista deja!");
      reusita = dirNou.mkdir();
      if (reusita)
        System.out.println("Am reusit sa cream directorul " +
          "C:\\dirProba");
      else
        System.out.println("Nu am creat directorul " +
          "C:\\dirProba!");
  }
}
```

5.3.2. Clasa **RandomAccessFile**

Această clasă se ocupă de lucrul cu fişiere binare (cu acces aleator direct), şi nu cu fluxurile fişierelor de caractere (cu acces secvenţial). Într-un fişier cu acces direct, putem căuta o dată de la o anumită poziţie, putem scrie sau citi date la o poziţie precizată, pe când într-un fişier cu acces secvenţial putem citi datele în ordinea în care apar. Clasa `java.io.RandomAccessFile` are doi constructori:

- `RandomAccessFile(String fisier, String mod);`
- `RandomAccessFile(File fisier, String mod);`

`String`-ul `mod` poate fi `"r"`, dacă se deschide fişierul în acces de citire, sau `"rw"`, dacă se deschide în acces de citire şi scriere. A doua formă de constructor este utilă atunci când există deja o instanţă a clasei `File`.

Exemplu 5.3.2. Înainte de a deschide un fişier în acces de citire/scriere, facem nişte verificări (dacă este fişier şi nu director, dacă se poate citi şi dacă se poate scrie în el):

```
File fisier = new File(cale);
if (! fisier.isFile() || ! fisier.canRead() ||
    ! fisier.canWrite()) {
  throw new IOException();
}
RandomAccessFile raf = new RandomAccessFile(fisier, "rw");
```

Metodele cele mai importante din această clasă sunt:

- `long getFilePointer() throws IOException;`
  Returnează poziţia curentă a pointerului din fişier (în octeţi) (returnează valoarea deplasamentului).
- `long length() throws IOException;`
  Returnează lungimea fişierului, în octeţi.

- `void seek(long pozitie) throws IOException;`
  Setează poziţia curentă din fişier (în octeţi). Fişierele încep cu poziţia `0`.

Metodele care suportă citirea şi scrierea de octeţi sunt:

- `int read() throws IOException;`
  Returnează următorul octet din fişier (memorat în octetul cel mai nesemnificativ al unei variabile de tip `int`) sau valoarea `-1`, dacă pointerul este la sfârşitul fişierului.
- `int read(byte destinatie[]) throws IOException;`
  Încearcă să citească suficienţi octeţi pentru a umple şirul `destinatie[]`. Returnează numărul de octeţi citiţi din fişier sau valoarea `-1`, dacă pointerul este la sfârşitul fişierului.
- `int read(byte destinatie[], int deplasament, int lungime) throws IOException;`
  Încearcă să citească `lungime` octeţi în şirul `destinatie[]`, începând de la `deplasament`. Returnează numărul de octeţi citiţi din fişier sau valoarea `-1`, dacă pointerul este la sfârşitul fişierului.
- `int write(int b) throws IOException;`
  Scrie octetul cel mai nesemnificativ al lui b.
- `int write(byte b[]) throws IOException;`
  Scrie toţi octeţii din şirul `b[]`.
- `int write(byte destinatie[], int deplasament, int lungime) throws IOException;`
  Scrie `lungime` octeţi din şirul `destinatie[]`, începând de la poziţia `deplasament`.

Clasa `RandomAccessFile` are metode pentru citirea şi scrierea tuturor tipurilor primitive de date. Aceste metode sunt:

- `boolean readBoolean();`
- `void writeBoolean(boolean b);`
- `byte readByte();`
- `void writeByte(int b);`
- `short readShort();`
- `void writeShort(int s);`
- `char readChar();`
- `void writeChar(int c);`
- `int readInt();`
- `void writeInt(int i);`
- `long readLong();`
- `void writeLong(long l);`
- `float readFloat();`
- `void writeFloat(float f);`
- `double readDouble();`
- `void writeDouble(double d);`
- `int readUnsignedByte();`

- `int readUnsignedShort();`
- `String readLine();`
- `String readUTF();`
- `void writeUTF(String s);`

Când un fişier cu acces aleator nu mai este folosit, atunci acesta trebuie închis prin apelul metodei:

- `void close() throws IOException;`

eliberându-se astfel resursele sistem de memorie asociate fişierului.

Exemplul 5.3.3. Fie programul Java de mai jos:

```
import java.io.*;

public class FisiereAccesAleator {
  public static void main(String args[]) {
    try {
      RandomAccessFile fisierUnu = new RandomAccessFile(
         "fisier.bin", "rw");
      fisierUnu.writeInt(10000);
      fisierUnu.writeInt(20000);
      fisierUnu.writeInt(30000);
      // vom inlocui in fisier valoarea 20000 cu 40000
      fisierUnu.seek(4);
      fisierUnu.writeInt(40000);
      fisierUnu.close();
    }
    catch (IOException e) {
      System.err.println("Eroare de scriere in fisierUnu " +
         e.toString());
    }
    try {
      // fisier.bin va contine valorile intregi 10000,
      // 40000 si 30000
      RandomAccessFile fisierDoi = new RandomAccessFile(
         "fisier.bin", "r");
      // citim primul intreg
      int i = fisierDoi.readInt();
      System.out.println("i1 = " + i + " pointer = " +
         fisierDoi.getFilePointer());
      // citim al doilea intreg
      i = fisierDoi.readInt();
      System.out.println("i2 = " + i + " pointer = " +
         fisierDoi.getFilePointer());
      // resetam pointerul catre fisierDoi la zero
```

```
        fisierDoi.seek(0);
        // citim patru octeti care formeaza un int
        byte b = fisierDoi.readByte();
        // afisam octetul cel mai semnificativ
        System.out.println("b4 = " + b);
        b = fisierDoi.readByte();
        System.out.println("b3 = " + b);
        b = fisierDoi.readByte();
        System.out.println("b2 = " + b);
        b = fisierDoi.readByte();
        // afisam octetul cel mai nesemnificativ
        System.out.println("b1 = " + b);
        // citim urmatorii opt octeti si îi interpretam ca double
        double d = fisierDoi.readDouble();
        System.err.println("d = " + d);
        fisierDoi.close();
      }
      catch (IOException e) {
        System.out.println("Eroare de citire din fisier " +
           e.toString());
      }
    }
  }
```

Mai întâi se va crea fişierul binar (prin acces de scriere) `fisier.bin`, în care sunt trei întregi: `10000`, `20000` şi `30000`. Apoi poziţionăm pointerul la fişier după patru octeţi (prin apelarea metodei `seek()`) şi astfel vom înlocui `20000` cu `40000`. Cu apelul metodei `getFilePointer()` vom verifica faptul că o variabilă de tip `int` se memorează într-un fişier binar pe patru octeţi. Apoi, poziţionăm pointerul la fişier pe prima poziţie şi citim pe rând primii patru octeţi (utilizând o variabilă de tip `byte`) şi apoi ultimii opt octeţi folosind o variabilă de tip `double`. Execuţia programului Java de mai sus va afişa:

```
i1 = 10000 pointer = 4
i2 = 40000 pointer = 8
b4 = 0
b3 = 0
b2 = 39
b1 = 16
d = 8.4879831653433E-310
```

5.3.3. Clase Java pentru fluxuri de intrare/ieşire

Clasele Java pentru fluxuri de intrare/ieşire consideră intrarea şi ieşirea ca secvenţe ordonate de octeţi. Uneori datele apar ca octeţi, întregi, numere reale etc. Clasele Java pentru fluxuri de intrare/ieşire pun la dispoziţie o abordare structurată asemănătoare celei din clasa `RandomAccessFile`. Acestea sunt:

- fluxuri de ieşire de nivel scăzut (inferior) care primesc şi scriu octeţi la un dispozitiv de ieşire;
- fluxuri de ieşire filtru de nivel înalt care primesc date în format general (primitive) şi scriu octeţi către un flux de ieşire de nivel jos sau către alt flux de ieşire filtru;
- fluxuri de scriere (engl. *writers*) sunt asemănătoare cu fluxurile de ieşire filtru specializate pe scriere de şiruri Java în unităţi de caractere Unicode;
- fluxuri de intrare de nivel scăzut care citesc octeţi de la un dispozitiv de intrare şi returnează octeţii apelantului;
- fluxuri de intrare filtru de nivel înalt care citesc date dintr-un flux de intrare de nivel jos sau de la un alt flux de intrare filtru şi returnează date în format general (primitive) către apelant;
- fluxurile de citire (engl. *readers*) sunt asemănătoare cu fluxurile de intrare filtru specializate pe citire de şiruri UTF în unităţi de caractere Unicode.

Java pune la dispoziţie pentru lucrul cu fluxuri clasele `InputStream/OutputStream` şi `Reader/Writer`. Primele clase lucrează cu fluxuri de octeţi (unitatea de transfer fiind de 8 biţi), în timp ce ultimele lucrează cu fluxuri de caractere, care în Java sunt reprezentate pe 16 biţi în format Unicode. Avantajul fluxurilor pe caractere Unicode este că oferă un suport conceptului de internaţionalizare. Acest concept permite dezvoltarea de aplicaţii Java care să se adapteze setărilor calculatorului pe care rulează, setări legate de limbă, zona geografică, ora locală etc.

În general, se obişnuieşte pentru fişiere utilizarea claselor `BufferedReader` şi `PrintWriter` (pentru citiri şi respectiv scrieri de linii de text), iar pentru intrări şi ieşiri care nu sunt propriu-zis fişiere (de exemplu: tastatură, pagini Web etc.) se foloseşte clasa `InputStreamReader`.

5.3.3.1. *Fluxuri de nivel scăzut*

Fluxurile de intrare de nivel scăzut au metode care citesc şi returnează intrarea ca octeţi. Fluxurile de ieşire de nivel înalt au metode care trimit octeţi şi scriu octeţi la ieşire. Vom începe cu prezentarea claselor `FileInputStream` şi `FileOutputStream`.

Cei mai utilizaţi constructori pentru fluxurile fişiere de intrare sunt:

- `FileInputStream(String cale);`
- `FileInputStream(File fisier);`

După ce un flux fişier de intrare a fost construit, putem apela metode de citire a unui octet, şir de octeţi, porţiuni de şiruri de octeţi (asemănător cu cele din clasa `RandomAccessFile`). Acestea sunt:

- `int read() throws IOException;`
  Returnează următorul octet din fişier (memorat în octetul cel mai nesemnificativ al unei variabile de tip `int`) sau `-1`, dacă pointerul este la sfârşitul fişierului.

- `int read(byte destinatie[]) throws IOException;`
  Încearcă să citească suficienţi octeţi pentru a umple şirul `destinatie[]`. Returnează numărul de octeţi citiţi din fişier sau `-1`, dacă pointerul este la sfârşitul fişierului.
- `int read(byte destinatie[], int deplasament, int lungime) throws IOException;`
  Încearcă să citească `lungime` octeţi în şirul `destinatie[]`, începând de la valoarea lui `deplasament`. Returnează numărul de octeţi citiţi din fişier sau `-1`, dacă pointerul este la sfârşitul fişierului.

Clasa `FileInputStream` conţine metodele utile:

- `int available() throws IOException;`
  Returnează numărul de octeţi care pot fi citiţi fără blocare.
- `void close() throws IOException;`
  Eliberează resursele sistem de memorie asociate fişierului. Un flux fişier de intrare trebuie închis mereu când nu mai este nevoie de el.
- `long skip(long numarOcteti) throws IOException;`
  Încearcă să citească şi să se deplaseze cu `numarOcteti` octeţi. Returnează numărul de octeţi cu care s-a deplasat în fişier.

Fluxurile fişiere de ieşire sunt aproape identice cu cele de intrare. Cei mai utilizaţi constructori pentru fluxurile fişiere de ieşire sunt:

- `FileOutputStream(String cale);`
- `FileOutputStream(File fisier);`
  Există metode pentru scrierea unui octet, şir de octeţi sau unui subşir de octeţi:
- `void write(int b) throws IOException;`
  Scrie octetul cel mai nesemnificativ al lui `b`.
- `void write(byte b[]) throws IOException;`
  Scrie toţi octeţii din şirul `b[]`.
- `void write(byte destinatie[], int deplasament, int lungime) throws IOException;`
  Scrie lungime octeţi din şirul `destinatie[]`, începând de la poziţia `deplasament`.

Clasa `FileOutputStream` oferă metoda `close()`, care trebuie apelată neapărat când nu mai este nevoie de fluxul fişier de ieşire.

Exemplul 5.3.4. Următorul program Java testează lucrul cu fluxuri fişiere `FileOutputStream`, respectiv `FileInputStream`:

```
import java.io.*;

public class FluxuriFisiere {
  public static void main(String args[]) {
    int i;
    try {
      FileOutputStream fisierUnu = new FileOutputStream(
          "fisier.txt");
      // declaram date de tip byte
      byte b = 5;
      // tipul byte = {-128, ..., 0, ..., 127}, deci necesita
      // un octet pentru memorare
      byte vectorUnu [] = {(byte)2, (byte)-128, (byte)127,
          (byte)0, (byte)6, (byte)-8};
      fisierUnu.write(b);
      fisierUnu.write(vectorUnu);
      fisierUnu.close();

      FileInputStream fisierDoi = new FileInputStream(
          "fisier.txt");
      int bInt = fisierDoi.read();
      // functia read() intoarce -1 daca am ajuns la sfarsitul
      // fisierului
      System.out.println(bInt + " = bInt ");
      System.out.println("disponibil = " +
          fisierDoi.available());
      b = (byte) fisierDoi.read();
      System.out.println(b + " = b ");
      b = (byte) fisierDoi.read();
      System.out.println(b + " = b ");
      byte vectorDoi[] = new byte[10];
      int spatiulRamas = fisierDoi.available();
      System.out.println("disponibil = " +
          fisierDoi.available());
      fisierDoi.read(vectorDoi);
      for (i = 0; i < spatiulRamas; i++)
        System.out.println("vectorDoi[" + i + "] = " +
            vectorDoi[i]);
      b = (byte) fisierDoi.read();
      // functia read() intoarce -1 daca am ajuns la
      // sfarsitul fisierului
      System.out.println(b + " ");
      fisierDoi.close();
    }
    catch (IOException e) {
      System.err.println("Eroare scriere fisier " + e);
    }
  }
}
```

Programul va afişa la execuţie:

```
5 = bInt
disponibil = 6
2 = b
-128 = b
disponibil = 4
vectorDoi[0] = 127
vectorDoi[1] = 0
vectorDoi[2] = 6
vectorDoi[3] = -8
-1
```

Pachetul `java.io` mai conţine şi alte clase pentru fluxurile, fişiere de intrare şi ieşire de nivel scăzut, printre care:

- `InputStream` şi `OutputStream`. Acestea sunt superclase pentru toate clasele de fluxuri de nivel jos. De exemplu, se pot utiliza pentru citirea şi scrierea socketurilor de reţea;
- `ByteArrayInputStream` şi `ByteArrayOutputStream`. Aceste clase citesc şi scriu şiruri de octeţi. Şirurile de octeţi nu sunt dispozitive de intrare/ieşire, dar clasele sunt utile pentru procesarea şi crearea secvenţelor de octeţi;
- `PipedInputStream` şi `PipedOutputStream`. Aceste clase furnizează un mecanism pentru comunicare sincronizată între fire de execuţie.

5.3.3.2. *Fluxuri filtru de nivel înalt*

Uneori este bine să citim din fluxuri informaţii de nivel înalt, cum ar fi întregi, `String`-uri etc. Java oferă suport pentru fluxuri de nivel înalt care se extind din superclasele `FilterInputStream` şi `FilterOutputStream`. Fluxurile de intrare de nivel înalt nu pot citi direct din dispozitive de intrare, cum ar fi fişiere sau socketuri. Ele citesc din alte fluxuri. Fluxurile de ieşire de nivel înalt nu pot scrie în dispozitive de ieşire, dar pot scrie în alte fluxuri.

Un exemplu de clasă des utilizată este `DataInputStream`, care are un singur constructor:

- `DataInputStream(InputStream fluxIntrare);`

Argumentul constructorului poate fi un flux fişier de intrare (`FileInputStream` extinde clasa `InputStream`), un flux de intrare dintr-un socket sau din alte tipuri de fluxuri de intrare. Când este apelată instanţa unui obiect `DataInputStream`, aceasta va realiza un număr de apeluri `read()` către `fluxIntrare`, să proceseze octeţii şi să returneze o valoare convenabilă. Cele mai utilizate metode ale clasei `DataInputStream` sunt:

- `boolean readBoolean() throws IOException;`
- `byte readByte() throws IOException;`
- `short readShort() throws IOException;`
- `char readChar() throws IOException;`
- `int readInt() throws IOException;`

- `long readLong() throws IOException;`
- `float readFloat() throws IOException;`
- `double readDouble() throws IOException;`
- `String readUTF() throws IOException;`
- `void close() throws IOException;`

Clasa `DataOutputStream` este asemănătoare cu clasa `DataInputStream` şi pune la dispoziţie constructorul:

- `DataOutputStream(OutputStream fluxIesire);`

Constructorul necesită primirea unui flux de ieşire. Când scriem într-un flux de ieşire, acesta converteşte parametrii metodei de scriere în octeţi şi îi scrie în flux `Iesire`. Cele mai utilizate metode ale clasei `DataOutputStream` sunt:

- `void writeBoolean(boolean b) throws IOException;`
- `void writeByte(int b) throws IOException;`
- `void writeBytes(String s) throws IOException;`
- `void writeShort(int s) throws IOException;`
- `void writeChar(int c) throws IOException;`
- `void writeInt(int i) throws IOException;`
- `void writeLong(long l) throws IOException;`
- `void writeFloat(float f) throws IOException;`
- `void writeDouble(double d) throws IOException;`
- `void writeUTF(String s) throws IOException;`
- `void close() throws IOException;`

Exemplul 5.3.5. Următorul program Java testează lucrul cu fluxuri filtru `DataOutputStream`, respectiv `DataInputStream`:

```
import java.io.*;

public class FluxuriFiltru {
  public static void main(String args[]) {
    try {
      FileOutputStream fisierUnu = new FileOutputStream(
         "fisierUnu.out");
      DataOutputStream fisierDateUnu = new DataOutputStream
      (fisierUnu);
      fisierDateUnu.writeByte(127);
      fisierDateUnu.writeInt(123456);
      fisierDateUnu.writeDouble(12.34);
      fisierDateUnu.writeUTF("123\u5555");
      fisierDateUnu.writeUTF(" Scriem acest text in fisier.");
      fisierDateUnu.writeInt(-1);
      fisierDateUnu.close();
      fisierUnu.close();
```

```
        System.out.println("Urmeaza citirea din fisier");
        // urmeaza citirea acestui fisier si afisarea datelor
        FileInputStream fisierDoi = new FileInputStream(
           "fisierUnu.out");
        DataInputStream fisierDateDoi = new DataInputStream(
           fisierDoi);
        byte b = fisierDateDoi.readByte();
        int i = fisierDateDoi.readInt();
        double d = fisierDateDoi.readDouble();
        String sirl = fisierDateDoi.readUTF();
        String sir2 = fisierDateDoi.readUTF();
        System.out.println("b = " + b);
        System.out.println("i = " + i);
        i = fisierDateDoi.readInt();
        System.out.println("i = " + i);
        System.out.println("d = " + d);
        System.out.println("sirl = " + sirl);
        System.out.println("sir2 = " + sir2);
        fisierDateDoi.close();
        fisierDoi.close();
      }
      catch (IOException e) {
        System.err.println("Eroare scriere fisier " + e);
      }
   }
}
```

Programul va afişa la execuţie:

```
Urmeaza citirea din fisier
b = 127
i = 123456
i = -1
d = 12.34
sirl = 123?
sir2 =  Scriem acest text in fisier.
```

Pachetul `java.io` oferă şi alte clase care se referă la fluxuri de nivel înalt. Constructorii fluxurilor de intrare de nivel înalt necesită ca argument un flux de nivel imediat mai jos. Acesta va fi sursa datelor citite de noul obiect. Similar, constructorii fluxurilor de ieşire de nivel înalt necesită un argument flux de nivel imediat mai jos. Noul obiect va scrie date către acest flux. Alte fluxuri de nivel înalt sunt:

- `BufferedInputStream` şi `BufferedOutputStream`: aceste clase au buffere interne, astfel încât octeţii pot fi citiţi sau scrişi în blocuri mari;
- `PrintStream`: această clasă este utilă la scrierea de text sau date primitive. Acestea sunt convertite la reprezentări caracter. Obiectele `System.out` şi `System.err` sunt instanţe ale acestei clase;

- `PushbackInputStream`: această clasă permite ca ultimul octet citit să fie pus înapoi în flux ca şi cum nu s-ar fi citit.

Exemplul 5.3.6. Putem implementa un flux de intrare care citeşte dintr-un flux de intrare de tip `buffer`, care la rândul lui citeşte dintr-un flux fişier de intrare, astfel:

```
FileInputStream fis = new FileInputStream("fisier.in");
BufferedInputStream bis = new BufferedInputStream(fis);
DataInputStream dis = new DataInputStream(bis);
```

5.3.3.3. *Fluxuri pentru citire şi fluxuri pentru scriere*

Fluxurile pentru citire (engl. *readers*) şi cele pentru scriere (engl. *writers*) sunt ca şi fluxurile de intrare şi ieşire: cele de nivel jos comunică cu dispozitivele de intrare/ ieşire, cele de nivel înalt comunică cu cele de nivel jos. Diferenţa dintre acestea este că fluxurile de citire şi de scriere utilizează numai caractere Unicode.

Clasa `FileReader` este un exemplu de cititor de nivel jos, având cei mai utilizaţi constructori:

- `FileReader(String cale);`
- `FileReader(File fisier);`

Scriitorul asociat acestei clase este `FileWriter`, având constructorii:

- `FileWriter(String cale);`
- `FileWriter(File fisier);`

Alte clase de cititori şi scriitori de nivel jos sunt:

- `CharArrayReader` şi `CharArrayWriter`: citesc şi scriu tablouri de caractere;
- `PipedReader` şi `PipedWriter`: furnizează un mecanism pentru comunicarea între procese;
- `StringReader` şi `StringWriter`: citesc şi scriu `String`-uri.

Toţi cititorii extind superclasa abstractă `Reader`. Această clasă oferă cele trei metode `read()` pentru citirea unui caracter, tablou de caractere sau a unui subşir de tablou de caractere. Atenţie! Unitatea de informaţie vehiculată este acum `char` (reprezentat pe doi octeţi) şi nu `byte` (cum se întâmpla la fluxurile de până acum):

- `int read() throws IOException;`
  Returnează următorul caracter din fişier (memorat în primii doi octeţi nesemnificativi ai unei variabile de tip `int`) sau `-1`, dacă pointerul este la sfârşitul fişierului.
- `int read(char destinatie[]) throws IOException;`
  Încearcă să citească suficiente caractere pentru a umple tabloul `destinatie[]`. Returnează numărul de caractere citite din fişier sau `-1`, dacă pointerul este la sfârşitul fişierului.
- `abstract int read(char destinatie[], int deplasament, int lungime) throws IOException;`

Încearcă să citească `lungime` caractere în tabloul `destinatie[]`, începând de la deplasament. Returnează numărul de caractere citite din fişier sau `-1`, dacă pointerul este la sfârşitul fişierului.

Toţi scriitorii extind superclasa abstractă `Writer`. Această clasă furnizează metode puţin diferite de cele trei bine cunoscute metode `write()`:

- `void write(int ch) throws IOException;`
  Scrie caracterul `ch` care apare în cei mai nesemnificativi doi octeţi ai lui `ch`.
- `void write(String str) throws IOException;`
  Scrie `String`-ul `str`.
- `void write(char c[]) throws IOException;`
  Scrie caracterele din şirul `c[]`.
- `void write(String str, int deplasament, int lungime) throws IOException;`
  Scrie `lungime` caractere din `String`-ul `str`, începând de la poziţia `deplasament`.
- `void write(char destinatie[], int deplasament, int lungime) throws IOException;`
  Scrie lungime caractere din tabloul `destinatie[]`, începând de la poziţia `deplasament`.

Cititorii şi scriitorii de nivel înalt se moştenesc din superclasele `Reader` sau `Writer`, deci au metodele listate mai sus. Ca şi fluxurile de nivel înalt, când se construieşte un cititor sau scriitor de nivel înalt, trebuie să trimitem un obiect din nivelul imediat următor. Clasele de nivel înalt sunt:

- `BufferedReader` şi `BufferedWriter`: aceste clase au *buffere* interne, astfel încât datele pot fi citite sau scrise în blocuri mari. Sunt similare cu cele de la fluxuri;
- `InputStreamReader` şi `OutputStreamReader`: aceste clase convertesc fluxuri de octeţi şi caractere Unicode. Implicit, clasele presupun că fluxurile folosesc codificarea implicită a caracterelor platformei;
- `LineNumberReader`: această clasă consideră intrarea ca fiind o secvenţă de linii de text. Apelul metodei `readLine()` returnează următoarea linie;
- `PrintWriter`: similară cu `PrintStream`, dar scrie caractere în loc de octeţi;
- `PushbackReader`: similară cu `PushbackInputStream`, dar citeşte caractere în loc de octeţi.

Exemplul 5.3.7. Următorul program pune în evidenţă utilizarea claselor `FileReader` şi `LineNumberReader` pentru afişarea unui fişier text, în care fiecare linie este precedată de numărul ei.

```
import java.io.*;

public class LiniiNumerotate {
  public static void main(String args[]) {
      try {
        FileReader fr = new FileReader("LiniiNumerotate.java");
        LineNumberReader lnr = new LineNumberReader(fr);
        String s;
        while ((s = lnr.readLine()) != null) {
          System.err.println(lnr.getLineNumber() + ": " + s);
        }
        lnr.close();
        fr.close();
      }
      catch (IOException e) {
        System.err.println(
          "A aparut o exceptie de intrare/iesire");
      }
    }
  }
```

După cum se observă, execuţia acestui program va afişa codul-sursă al programului de mai sus, punând la începutul fiecărei linii numărul ei, urmat de caracterul două puncte şi spaţiu.

Continuăm cu prezentarea în detaliu a două clase frecvent utilizate.

5.3.3.4. *Clasele* `PrintWriter` *şi* `BufferedReader`

Clasa `PrintWriter` are patru constructori:

- `public PrintWriter(Writer iesire);`
  Creează un nou obiect `PrintWriter`, fără trimiterea automată a conţinutului unei linii. Parametrul `iesire` este un flux orientat caracter.
- `public PrintWriter(Writer iesire, boolean autoFlush);`
  Creează un nou obiect `PrintWriter`, cu trimiterea automată a conţinutului unei linii (în cazul metodelor `println()`), dacă variabila `autoFlush` are valoarea `true`. Parametrul `iesire` este un flux orientat caracter.
- `public PrintWriter(OutputStream iesire);`
  Creează un nou obiect `PrintWriter`, fără trimiterea automată a conţinutului unei linii. Parametrul `iesire` este un flux de ieşire. Constructorul este util deoarece creează un obiect intermediar `OutputStreamWriter`, care va converti caracterele în octeţi folosind codificarea implicită a caracterelor.
- `public PrintWriter(OutputStream iesire, boolean autoFlush);`
  Creează un nou obiect `PrintWriter`, cu trimiterea automată a conţinutului unei linii (în cazul metodelor `println()`). Parametrul `iesire` este un flux de ieşire. Constructorul este util deoarece creează un obiect intermediar `OutputStreamWriter`, care va converti caracterele în octeţi folosind codificarea implicită a caracterelor.

Clasa `BufferedReader` are doi constructori:

- `public BufferedReader(Reader intrare, int marime);`

Creează un flux de intrare caracter asociat cititorului `intrare` care utilizează un buffer de dimensiune egală cu `marime`. Dacă valoarea lui `marime` este `0`, atunci se aruncă o excepţie `IllegalArgumentException`.

- `public BufferedReader(Reader intrare);`
  Creează un flux de intrare caracter asociat cititorului `intrare` care utilizează un buffer de dimensiune implicită.

Pentru citirea şi scrierea de linii de text, se pot utiliza metodele:

- `readLine()` din clasa `BufferedReader`. Aceasta citeşte o linie de text într-un şir (dacă dorim împărţirea acestui şir, folosim clasa `StringTokenizer`);
- `print()` şi `println()` din clasa `PrintWriter`. Aceste metode scriu un şir într-un fişier (`println()` adaugă la sfârşitul fiecărei date scrise şi caracterul `'\n'`).

Exemplul 5.3.8. Vom prezenta un program Java de sine stătător care ne permite să scriem caractere într-o arie text şi apoi să salvăm textul într-un fişier. Vom declara o variabilă auxiliară pentru lucrul cu fişierul de ieşire:

```
PrintWriter fisierIesire = null;
```

Apoi creăm o instanţă a clasei `PrintWriter` astfel:

```
fisierIntrare = new PrintWriter(
   new FileWriter("fisier.txt"), true);
```

Dacă fişierul `fisier.txt` nu există, atunci va fi creat. Dacă există, atunci va fi suprascris. S-a apelat constructorul clasei `FileWriter`, care permite specificarea numelui fişierului ca un şir de caractere. Acesta este transmis ca prim argument constructorului clasei `PrintWriter`. Cel de-al doilea parametru este o valoare booleană. Dacă este `true`, atunci bufferul intern va fi automat trimis (engl. *flushed*) în fişier la fiecare apel al metodei `print()`. În caz contrar, trebuie apelată explicit metoda `flush()`. În plus, dacă o parte ulterioară a execuţiei programului se blochează, atunci conţinutul din buffer este pierdut. Pentru a scrie în fişier folosim metoda `print()`:

```
fisierIesire.print(intrareArieText.getText());
```

În final, închidem fişierul astfel:

```
fisierIesire.close();
```

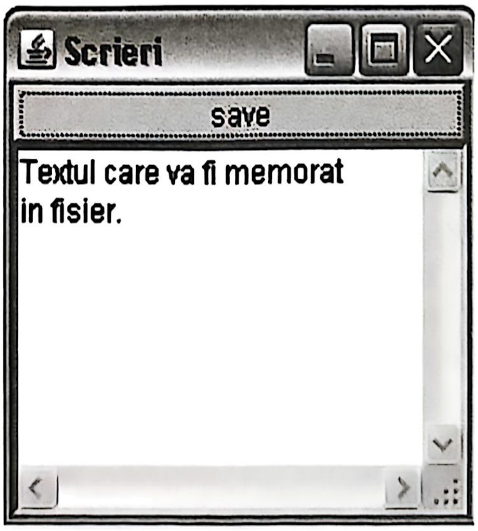
Crearea unui fişier poate produce o excepţie, deci aceasta trebuie verificată, fiind obligatorie „înconjurarea" codului într-un bloc `try-catch` pentru detectarea unei eventuale excepţii de tip `IOException`. O verificare şi mai riguroasă a posibilelor erori este cazul în care metoda `print()` detectează o eroare în timp ce scrie într-un fişier:

```
fisierIesire.print(intrareArieText.getText());
if (fisierIesire.checkError())
   System.err.println("Eroare in timpul scrierii in fisier");
```

Codul complet al aplicaţiei discutate mai sus este următorul:

```
import java.io.*;
// importarea pachetelor pentru lucru cu obiecte grafice
import java.awt.*;
import java.awt.event.*;

public class Scrieri extends Frame implements
     ActionListener {
  private TextArea intrareArieText;
  private Button butonSalvare;

  /** Constructorul clasei
    * Stabileste titlul ferestrei.*/
  public Scrieri() {
    super("Scrieri");
  }

  public static void main(String[] args) {
    Scrieri f = new Scrieri();
    f.setSize(200, 200);
    f.interfataGrafica();
    f.setVisible(true);
  }
  // crearea elementelor grafice si
  // adaugarea acestora la fereastra aplicatiei
  public void interfataGrafica() {
    butonSalvare = new Button("save");
    add("North", butonSalvare);
    butonSalvare.addActionListener(this);
    intrareArieText = new TextArea(10, 50);
    add("Center", intrareArieText);
    // ascultator pentru inchiderea ferestrei
    addWindowListener(new WindowAdapter(){
      // aceasta metoda este apelata automat cand se apasa
      // butonul pentru inchiderea ferestrei aplicatiei
      public void windowClosing(WindowEvent event) {
        System.exit(0);
      }
    });
  }

  // aceasta metoda este apelata cand s-a apasat
  // butonul 'save'
  public void actionPerformed(ActionEvent event) {
      if (event.getSource() == butonSalvare) {
        PrintWriter fisierIesire = null;
        try {
          // salvarea textului in fisier
          fisierIesire = new PrintWriter(new FileWriter(
            "fisier.txt"), true);
          fisierIesire.println(intrareArieText.getText());

        }
        catch (IOException e) {
          System.err.println("Eroare de fisier: " +
             e.toString());
          System.exit(1);
        }
        finally {
          if (fisierIesire != null)
             fisierIesire.close();
        }
      }
    }
  }
```

Execuţia programului Java de mai sus va implica apariţia cadrului următor. După completarea ariei text, se poate apăsa butonul „`save`" şi are loc salvarea textului în fişierul „`fisier.txt`".



Textul care va fi memorat
in fisier.

Fereastra aplicaţiei pentru salvare în fişier

Exemplul 5.3.9. Vom prezenta un program Java de sine stătător care ne permite să citim un fişier pentru a fi încărcat într-o arie text. La început, declarăm o instanţă:

```
BufferedReader fisierIntrare = null;
```

Creăm apoi o instanţă:

```
fisierIntrare = new BuffererReader(new FileReader(numeFisier));
```

Apoi citim câte o linie din fişier, adăugând-o în aria text (utilizăm metoda `append()`). Apelăm metoda `readLine()`, care întoarce `null` dacă s-a ajuns la sfârşitul fişierului:

```
while ( (linie = fisierIntrare.readLine()) != null ) {
  arieTextIntrare.append(linie + "\n");
}
fisierIntrare.close();
```

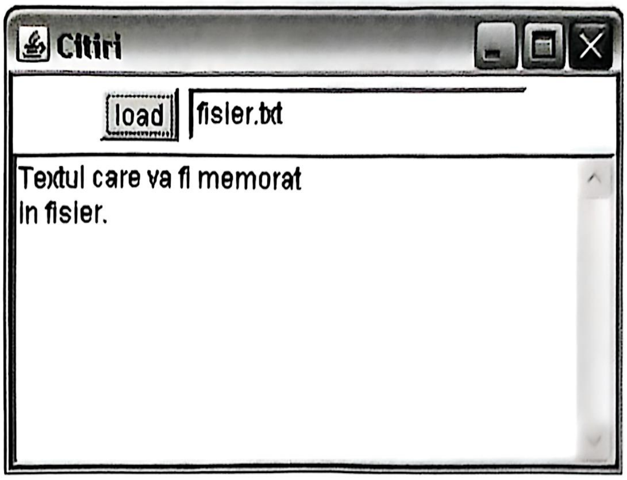
Când utilizatorul apasă butonul `load`, programul va executa următoarele acţiuni:

1. citeşte numele fişierului din câmpul text;
2. deschide fişierul cu acest nume;
3. citeşte linii din fişier şi le adaugă la aria text, până la sfârşitul fişierului;
4. închide fişierul.

Codul complet al programului Java discutat este:

```
import java.io.*;
import java.awt.*;
import java.awt.event.*;

public class Citiri extends Frame implements
  ActionListener {
  private TextArea arieTextIntrare;
  private Button butonLoad;
  private TextField numeCamp;

  public static void main(String[] args) {
    Citiri f = new Citiri();
    f.setBounds(200, 100, 300, 200);
    f.interfataGrafica();
    f.setTitle("Citiri");
    f.setVisible(true);
  }

  // crearea elementelor grafice si
  // adaugarea acestora la fereastra aplicatie
  public void interfataGrafica() {
    Panel top = new Panel();
    butonLoad = new Button("load");
    top.add(butonLoad);
    butonLoad.addActionListener(this);
    numeCamp = new TextField(20);
    top.add(numeCamp);
    numeCamp.addActionListener(this);
    add("North", top);
    arieTextIntrare = new TextArea("", 10, 50);
    add("Center", arieTextIntrare);
    // stabilim un ascultator pentru inchiderea ferestrei
    addWindowListener(new WindowAdapter(){
        // aceasta metoda se va apela la apasarea butonului
        // 'Close' pentru inchiderea ferestrei aplicatiei
        public void windowClosing(WindowEvent event) {
          System.exit(0);
        }
    });
  }

  // metoda se va apela la apasarea butonului 'load'
  public void actionPerformed(ActionEvent event) {
    if (event.getSource() == butonLoad) {
      String numeFisier;
      numeFisier = numeCamp.getText();
      BufferedReader fisierIntrare = null;
      try {
        fisierIntrare = new BufferedReader(new FileReader
        (numeFisier));
        arieTextIntrare.setText(""); // stergem continutul
        //ariei text
        String line;
        // citim cate o linie din fisier si o adaugam la aria
        // text
        while ((line = fisierIntrare.readLine()) != null) {
          arieTextIntrare.append(line + "\n");
        }
        fisierIntrare.close();
      }
      catch (IOException e) {
       System.err.println("Eroare in fisierul: " + numeFisier +
          ": " + e.toString());
      }
    }
  }
}
```

Iată o posibilă execuţie a acestui program Java:



fisier.txt

Textul care va fi memorat
in fisier.

Fereastra aplicaţiei pentru citire din fişier

5.3.3.5. *Clasa* `StreamTokenizer`

Clasa `java.io.StreamTokenizer` oferă suport pentru parsarea datelor dintr-un flux de intrare. Aceasta extrage câte o entitate (engl. *token*) şi poate recunoaşte valorile numerice, şiruri de caractere date între simbolurile ghilimele, identificatori şi comentarii.

Clasa amintită are un singur constructor:

- `public StreamTokenizer(Reader r)`

unde `r` este fluxul de intrare de tip `Reader` care va fi parsat.

Pentru uşurinţa identificării elementelor şi lucrul cu acestea, clasa `StreamTokenizer` pune la dispoziţie următoarele date membre:

Denumirea atributului	Descrierea atributului
`public double` **`nval`**	Dacă elementul curent este număr, atunci valoarea acestuia se va regăsi în acest atribut.
`public String` **`sval`**	Dacă elementul curent este de tip şir de caractere, atunci şirul corespunzător va fi stocat în acestă dată membră.
`public static final int` **`TT_EOF`**	Constantă care indică faptul că fluxul de intrare a ajuns la sfârşit.
`public static final int` **`TT_EOL`**	Constanta arată că s-a citit sfârşitul unei linii.
`public static final int` **`TT_NUMBER`**	Constanta indică întâlnirea unui număr.
`public static final int` **`TT_WORD`**	Constanta semnalează citirea unui cuvânt.
`public int` **`ttype`**	Indică tipul de element depistat şi poate lua ca valoare una din constantele prezentate mai sus

Pentru a vedea utilitatea clasei `StreamTokenizer` vom urmări facilităţile date de metodele acesteia:

Denumirea metodei	Descrierea metodei
public void **wordChars**(int min, int max)	Stabileşte intervalul în care se găsesc caracterele constituente ale cuvintelor.
public void **whitespaceChars**(int min, int max)	Indică intervalul în care sunt cuprinse spaţiile albe. Spaţiile albe nu vor fi luate în considerare.
public void **ordinaryChars**(int min, int max)	Stabileşte intervalul pentru caracterele ordinare (cele care sunt considerate token-uri). Când se întâlneşte un astfel de caracter, va fi considerat de tip şir de caractere.
public void **commentChar**(int c)	Stabileşte caracterul care indică începutul unui comentariu ce ţine până la sfârşitul liniei. Comentariile sunt ignorate.
public void **quoteChar**(int c)	Fixează caracterul utilizat pentru încadrarea şirurilor de caractere.
public void **parseNumbers**()	Indică faptul că numerele vor fi identificate, iar valoarea lor se va regăsi în atributul `nval`.
public void **eolIsSignificant**(boolean flag)	Stabileşte dacă sfârşitul de linie prezintă sau nu importanţă. În caz afirmativ, la întâlnirea acestuia atributul `ttype` va lua valoarea constantei `TT_EOL`.
public void **slashStarComments**(boolean flag)	Specifică dacă textul cuprins între `/*` şi `*/` este considerat comentariu.
public void **slashSlashComments**(boolean flag)	Specifică dacă textul cuprins între `//` şi sfârşit de linie este considerat comentariu.
public void **lowerCaseMode**(boolean fl)	Specifică dacă textul va fi convertit automat la minuscule.
public int **nextToken**() throws IOException	Citeşte următorul element (token) şi îi identifică tipul. Returnează valoarea câmpului `ttype`, iar în cazul în care apar probleme la citirea din flux, va fi aruncată o excepţie de tip `IOException`.
public void **pushBack**()	Pune înapoi elementul citit în flux (la început) fără a modifica valorile atributelor.
public int **lineno**()	Returnează numărul liniei curente.
public String **toString**()	Returnează şirul de caractere corespunzător elementului curent.

Exemplul 5.3.10. Programul Java va citi datele din fişierul `fisier.txt`. Acesta are următorul conţinut:

```
# comentariu
Ioana are 125.75 EURO.
```

```
/* comentariu
pe mai multe randuri */
```

Se vor parsa datele şi vor fi identificate numerele şi cuvintele. Codul programului este următorul:

```
import java.io.*;
import static java.io.StreamTokenizer.*;

public class Identificare {
   static public void main(String arg[]) {
      StreamTokenizer parser;
      try {
         /* crearea unui flux de intrare dintr-un fisier
            care va fi parsat de un StreamTokenizer */
         parser = new StreamTokenizer(new FileReader(
           "fisier.txt"));
         /* Caracterul '#' va stabili un comentariu
            pana la sfarsitul liniei */
         parser.commentChar('#');
         // Vor fi permise comentariile asemenea acestuia
         parser.slashStarComments(true);
         // Sfarsitul de linie va fi ignorat
         parser.eolIsSignificant(false);
         // citirea din flux si parsarea acestuia
         while(parser.nextToken() != TT_EOF)
           // daca am intalnit un numar
           if (parser.ttype == TT_NUMBER)
             System.out.println("Numar: " + parser.nval);
           // daca s-a identificat un sir de caractere
           else if (parser.ttype == TT_WORD)
             System.out.println("Cuvant: " + parser.sval);
           // daca am intalnit sfarsitul unei linii
           else if (parser.ttype == TT_EOL)
             /* mai sus am stabilit ignorarea sfarsiturilor de
             linie */
             System.out.println("Sfarsit de linie");
      } // tratarea exceptiei
      catch(IOException e) {
         System.err.println("Eroare de intrare/iesire: " + e);
      }
   }
}
```

Pentru simplitatea codului am importat membrii statici ai clasei `StreamTokenizer`. Astfel nu a trebuit să indicăm clasa din care fac parte pentru membrii statici `TT_EOF`, `TT_NUMBER`, `TT_WORD` şi `TT_EOL`.

Execuţia programului anterior va afişa la consolă:

```
Cuvant: Ioana
Cuvant: are
Numar: 125.75
Cuvant: EURO.
```

Observăm că valoarea numerică a fost identificată, iar comentariile şi sfârşiturile de linie au fost ignorate.

Exemplul 5.3.11. Vom prezenta un program Java de sine stătător care ne permite să căutăm anumite informaţii în fişier. Presupunem că avem un fişier cu linii de forma:

```
nume, nota1, nota2 <SfarsitLinie>
```

Din punct de vedere al interfeţei grafice, avem cinci câmpuri text, unul de afişare a mesajului de citire a numelui fişierului, două câmpuri editabile, unul pentru citirea numelui fişierului şi celălalt pentru citirea numelui persoanei ale cărei note trebuie căutate în fişierul deschis pentru citire. Mai există două câmpuri text pentru afişarea celor două note găsite. Începerea căutării se semnalează ca eveniment când se apelează butonul `Cautare`. Se presupune că numele persoanei este unic în baza de date. Spre deosebire de programul precedent, trebuie să procesăm linia curentă folosind un obiect de tip `java.util.StringTokenizer`. La parcurgerea fişierului se procedează aproximativ astfel (pseudocod):

```
boolean gasit = false;
while ( (exista linii de citit) && (not gasit)) {
  citeste inregistrare;
  if (primul camp coincide cu numele) {
    gasit = true;
    afiseaza notele in ariile text
  }
}
```

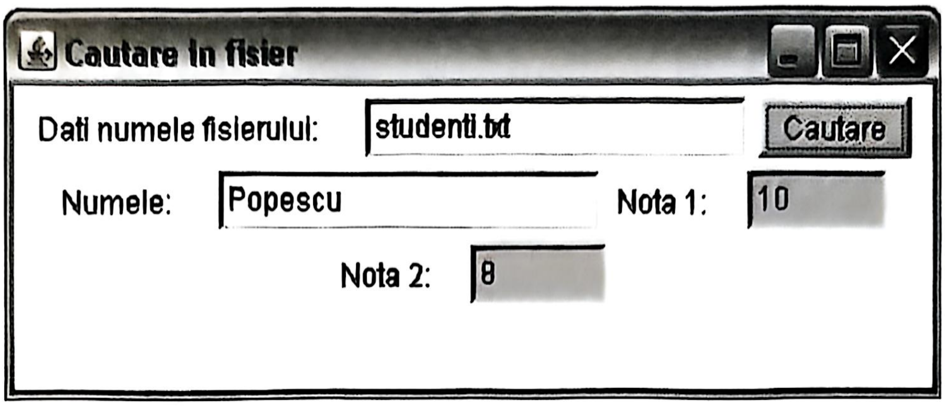
Codul complet al programului Java asociat este:

```
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class CautareFisier extends Frame implements
    ActionListener {
  // declararea datelor membre
  private Button butonCautare;
  private TextField rezultat1;
  private TextField rezultat2;
  private TextField campPersoana;
private TextField campNumeFisier;

public static void main(String[] args) {
  CautareFisier f = new CautareFisier();
  f.setBounds(200, 100, 400, 150);
  f.interfataGrafica();
  f.setTitle("Cautare in fisier");
  f.setVisible(true);
 }

public void interfataGrafica() {
  setLayout(new FlowLayout());
  Label eticheta  = new Label("Dati numele fisierului: ");
  add(eticheta);
  campNumeFisier = new TextField(20);
  campNumeFisier.setText("");
  add(campNumeFisier);
  butonCautare = new Button(" Cautare ");
  add(butonCautare);
  butonCautare.addActionListener(this);
  add(new Label("Numele: "));
  campPersoana = new TextField(20);
  campPersoana.setText("");
  add(campPersoana);
  add(new Label("Nota 1:"));
  rezultat1 = new TextField(5);
  rezultat1.setEditable(false);
  add(rezultat1);
  add(new Label("Nota 2:"));
  rezultat2 = new TextField(5);
  rezultat2.setEditable(false);
  add(rezultat2);
  // pentru inchiderea ferestrei
  this.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent ev) {
      System.exit(0);
    }
  });
}

public void actionPerformed(ActionEvent ev) {
  if (ev.getSource() == butonCautare) {
    String numeFisier = campNumeFisier.getText();
    BufferedReader fisierIntrare = null;
    try {
        fisierIntrare = new BufferedReader(
                  new FileReader(numeFisier));
      }
      catch (IOException e) {
       System.err.println("Nu gasim fisierul: " + numeFisier +
           ": " + e.toString());
        return;
      }
      try {
        String linie;
        boolean gasit = false;
        while (((linie = fisierIntrare.readLine()) != null) &&
        (! gasit)) {
          StringTokenizer tokens = new StringTokenizer(
            linie, " ,");
          String numeFisierIntrare = tokens.nextToken();
          if (campPersoana.getText().equals(
            numeFisierIntrare)) {
            gasit = true;
            rezultat1.setText(tokens.nextToken());
            rezultat2.setText(tokens.nextToken());
          }
        }
        fisierIntrare.close();
      }
      catch (IOException e) {
       System.err.println("Eroare in fisierul: " + numeFisier +
           ": " + e.toString());
        System.exit(1);
      }
    }
  }
}
```

Presupunem că avem fişierul `studenti.txt`, astfel:

```
Popescu, 10, 8
Ionescu, 6, 7
Pop, 9, 7
```

Atunci o execuţie a programului de mai sus poate fi:



Dati numele fisierului: studenti.txt

Numele: Popescu

Nota 1: 10

Nota 2: 8

Fereastra aplicaţiei pentru căutare în fişier

Atenţie! A nu se face confuzie între clasele `java.io.StreamTokenizer`, care parsează un flux de intrare, şi `java.util.StringTokenizer`, care parsează un şir de caractere. În plus, clasa `StreamTokenizer` oferă suport pentru identificarea valorilor numerice, ignorarea comentariilor etc.

5.3.3.6. *Clasa* `System`

Această clasă pune la dispoziţie trei fluxuri predefinite: `System.in`, `System.out` şi `System.err`. Aceste fluxuri sunt cunoscute sub denumirea de ecran *terminal* sau *consola* de intrare/ieşire. Fluxul `System.in` poate fi folosit direct pentru citirile de la tastatură şi pentru citirile redirectate din comenzi ale sistemului de operare. Totuşi, metodele puse la dispoziţie sunt de nivel jos, spre deosebire de citirea clasică dintr-un şir de caractere. Recomandăm utilizarea constructorului `BufferedReader()` şi metoda `readLine()`, astfel:

```
BufferedReader tastatura =
     new BufferedReader(new InpuStreamReader(System.in), 1);
```

Spre deosebire de apelurile precedente ale constructorului `BufferedReader`, acum avem doi parametri, al doilea fixând lungimea bufferului la `1`, eliminând astfel „efectul de buffer". Constructorul `BufferedReader` (fără precizarea lungimii bufferului) aşteaptă umplerea bufferului său (cu mai multe linii de text). Doar când bufferul este plin se va transmite prima linie către `readLine()`. Efectul este că utilizatorul trebuie să introducă mai multe linii de text pentru ca prima linie să fie accesată de către program. De aceea am setat lungimea bufferului la `1`. Astfel `readLine()` va citi textul introdus imediat după tastarea lui `<ENTER>`. Odată creat un flux, citirea din el se realizează astfel:

```
String linie = tastatura.readLine();
```

Exemplul 5.3.12. Vom prezenta un program Java de sine stătător care ne permite să citim un `String` de la tastatură.

```
import java.io.*;

public class CitiriTastatura {
  public static void main(String[] args) {
    CitiriTastatura obiect = new CitiriTastatura ();
    obiect.executa();
  }

  private void executa()  {
    BufferedReader tastatura = new BufferedReader(
      new InputStreamReader(System.in), 1);
    System.out.print("Dati un sir: ");
    try {
      System.out.flush(); // goleste bufferul de iesire la
                          // consola
      String linie = tastatura.readLine();
      System.out.println("Ati tastat " + linie);
    }
    catch (IOException e) {
      System.err.println("Intrare de la tastatura " +
         e.toString());
      System.exit(2);
    }
  }
}
```

Fluxul `System.out` are la dispoziţie metodele `print()`, `println()` pentru afişarea unui `String`. Comanda `System.out.flush()` asigură că textul care a fost argument pentru `print()` va fi imediat afişat (nu este şi cazul lui `println()`, unde nu este nevoie de `flush()`). Fluxul `System.err` poate fi utilizat similar cu `print()` şi `println()`. Mesajele de eroare apar pe ecranul consolei. De exemplu, putem avea un apel de genul:

```
System.err.println("Eroare! Fisierul nu a fost gasit!");
```

Clasa `System` pune la dispoziţie o metodă pentru terminarea imediată a aplicaţiilor. Este vorba despre `System.exit()`. Această funcţie are un argument întreg. Convenţia de ieşire normală este 0, în timp ce numere diferite de 0 înseamnă că a avut loc o eroare şi programul nu s-a terminal normal. De exemplu, putem avea mesajele:

```
System.exit(0); //iesire cu succes
System.exit(3); //iesire cu cod eroare 3
```

Codul întors la terminarea execuţiei unui program poate fi utilizat pentru a determina dacă acesta s-a executat cu succes sau pentru a stabili natura erorii apărute.

Exemplul 5.3.13. Vom prezenta un program Java de sine stătător care permite căutarea unui `String` într-un fişier. Numele fişierului din care se face citirea va fi furnizat ca argument în linia de comandă şi memorat în `args[0]` din metoda `main(String args[])`. Considerăm că fişierul-sursă al aplicaţiei care realizează căutarea se numeşte `Gaseste.java`. Atunci comanda de execuţie a aplicaţiei poate fi:

```
java Gaseste fis.in
```

unde `fis.in` este numele fişierului de intrare. Apoi citim de la tastatură cuvântul pe care dorim să-l căutăm în respectivul fişier. Vom obţine liniile care conţin acel cuvânt, plus o linie precedentă şi una ulterioară acelei linii în caz afirmativ. Programul afişează toate apariţiile cuvântului. În cazul în care cuvântul nu este găsit nu se va afişa nimic. Codul complet al programului Java care implementează cele discutate mai sus este:

```
import java.io.*;

class Gaseste {
  private String linia1, linia2, linia3;
  private BufferedReader tastatura, fluxIntrare;

  public static void main(String[] args) {
    Gaseste gaseste = new Gaseste();
    gaseste.cauta(args[0]);
  }

  // metoda care realizeaza cautarea efectiva
  private void cauta(String numeFisier) {
    tastatura = new BufferedReader(new InputStreamReader(
      System.in), 1);
    /* se apeleaza metoda prompt() care intoarce sirul citit
      de la tastaura */
    String sir = prompt("Dati sirul ce trebuie cautat: ");
    linia1 = "";
    linia2 = "";
    try {
      fluxIntrare = new BufferedReader(new FileReader(
        numeFisier));
      /* in linia3 se va citi cate o linie din fisierul de
        intrare */
      while ((linia3 = fluxIntrare.readLine()) != null) {
        /* daca linia2 contine sirul citit de la tastatura
          se va afisa linia in care apare (linia2) si
          contextul (o linie inainte si una dupa) */
        if (linia2.indexOf(sir) >= 0)
          afiseazaLinia();
        // se pastreaza ultimele doua linii citite
        linia1 = linia2;
        linia2 = linia3;
      }
      // am ajuns la sfarsitul fisierului
      linia3 = "";
      // testam daca sirul cautat apare pe ultima linie
      if (linia2.indexOf(sir) >= 0)
        afiseazaLinia();
      // inchidem fisierul de intrare
      fluxIntrare.close();
    }
    catch (IOException e) {
      System.err.println("Eroare la cautarea sirului: " +
        e.toString());
      System.exit(1);
    }
  }

  // aceasta metoda afiseaza linia gasita si contextul in care
  // apare
  private void afiseazaLinia() {
    System.out.println("<<-- context:");
    System.out.println(linia1);
    System.out.println(linia2);
    System.out.println(linia3);
    System.out.println("                       -->>");
    System.out.println("");
  }

  /* aceasta metoda afiseaza sirul de caractere primit (mesaj)
    si citeste un sir de caractere de la tastatura */
  private String prompt(String mesaj) {
    String raspuns = "";
    try {
      System.out.print(mesaj);
      System.out.flush();
      raspuns = tastatura.readLine();
    }
    catch (IOException e) {
      System.err.println("Eroare la citirea de la tastatura: "
        + e.toString());
      System.exit(2);
    }
    return raspuns;
  }
}
```

5.4. Serializarea obiectelor

O caracteristică importantă a aplicaţiilor în reţele este necesitatea de a trimite şi primi date între două aplicaţii, posibil între maşini aflate la distanţă, sau salvarea stării curente a obiectelor şi restaurarea acestora la o execuţie ulterioară. Acest lucru se realizează în trei faze:

1. împachetarea datelor (*serializare*);
2. trimiterea/salvarea datelor;
3. despachetarea datelor (*deserializare*).

1. *Serializarea* este procesul de împărţire a datelor în octeţi, astfel încât să poată fi trimişi printr-un flux de ieşire. Utilizând fluxuri normale, un întreg se transmite împărţindu-l în patru octeţi şi scriindu-l într-un obiect `OutputStream`. Serializarea datelor de tipuri primitive se realizează cu ajutorul clasei `DataOutputStream`. Dacă dorim să trimitem date complexe reprezentând un obiect, trebuie să împărţim obiectul într-o formă astfel încât să poată fi reconstruit ulterior. Dacă obiectul referă alte obiecte, atunci procesul poate fi complex. Se restaurează o copie a obiectului original şi o listă de obiecte care-l referă, chiar dacă maşina virtuală este aceeaşi. Cu toate acestea, dacă un obiect este transmis de două ori în acelaşi flux, atunci va reface două referinţe la aceeaşi copie. Obiectul serializabil implementează o interfaţă pentru a rezolva aceasta la un nivel înalt, preluând din sarcina programatorului serializarea obiectelor (permiţând totuşi flexibilitatea trimiterii obiectelor).
2. Trimiterea/salvarea datelor se referă la procesul ajungerii datelor de la sursă la destinaţie, respectiv din program într-un fişier extern. Fluxurile obiect Java furnizează facilităţi de serializare a obiectelor de-a lungul fluxurilor orientate octet, deci se pot accesa toate mecanismele de transport existente.
3. *Deserializarea* se referă la despachetarea datelor (procesul invers al serializării). Serializarea reduce datele într-un şir de octeţi, dar păstrează suficiente informaţii pentru reconstrucţia ulterioară a datelor. Serializarea şi deserializarea se realizează în Java cu ajutorul claselor `ObjectOutputStream` şi `ObjectInputStream`.

5.4.1. Clasa `ObjectOutputStream`

Clasa `ObjectOutputStream` implementează interfaţa `ObjectOutput` (derivată din interfaţa `DataOutput`) şi are toate metodele definite (nici una nu este abstractă). Asemenea unui `FilterOutputStream`, acesta se ataşează unui `OutputStream` existent în maniera cunoscută.

Clasa `ObjectOutputStream` are doar un constructor:

- `ObjectOutputStream(OutputStream iesire) throws IOException;`
  Acesta creează un obiect `ObjectOutputStream` ataşat argumentului `iesire`.

Constructorul poate arunca o excepţie `IOException`, deoarece în fluxul ataşat se scrie imediat un antet (engl. *header*).

Exemplul 5.4.1. După cele discutate mai sus, putem utiliza constructorul clasei `ObjectOutputStream` astfel:

```
FileOutputStream unFlux = new FileOutputStream("t.tmp");
ObjectOutputStream oos = new ObjectOutputStream(unFlux);
```

Când sunt serializate tipuri primitive, atunci se pot utiliza metode ale clasei `DataOutput`. Dintre metodele clasei `ObjectOutputStream` le enumerăm pe cele mai importante:

- `void writeObject(Object o) throws IOException;`
  Este responsabilă de serializarea obiectelor şi trimiterea lor către flux.

Exemplul 5.4.2. Folosind notaţiile din exemplele de mai sus, avem:

```
int [] sir = new int[10];
oos.writeObject(sir);
Color culoare = new Color(240, 240, 240);
oos.writeObject(culoare);
```

Metoda `writeObject()` poate trimite toate câmpurile unui obiect, inclusiv câmpurile `private` şi `protected`. Doar câmpurile marcate `transient` sau `static` nu vor fi trimise. Din cauza acestor implicaţii de securitate, o clasă trebuie să specifice explicit permisiunea de a fi serializată prin implementarea interfeţei `Serializable` sau `Externalizable`. Încercarea de a serializa un obiect care nu implementează una dintre aceste interfeţe va implica o excepţie `NotSerializableException`.

- `void reset() throws IOException;`
  Resetează obiectul `ObjectOutputStream` către o stare iniţială, ca atunci când a fost creat. Fiecare obiect `ObjectOutputStream` păstrează urma fiecărui obiect (din clasa `Object`) care a fost trimis. Astfel, dacă acelaşi obiect este scris către un `ObjectOutputStream` de două ori, atunci a doua oară se va scrie o referinţă înapoi, chiar dacă obiectul original a fost modificat. Acest lucru este important, deoarece chiar dacă au fost făcute referinţe între obiecte de-a lungul apelurilor metodei `writeObject()`, se va putea reconstitui structura de referinţe iniţială. După un apel al lui `reset()`, dacă un obiect se va retrimite, atunci se va trimite o nouă copie.
- `void defaultWriteObject() throws IOException;`
  Scrie obiectul apelant (obiectul care apelează aceasta metodă) către un `ObjectOutputStream`. Această metodă este utilizată doar în implementările obişnuite ale lui `writeObject()`.
- `protected Object replaceObject(Object obiect) throws IOException;`
  Este utilizată de subclasele `ObjectOutputStream` pentru înlocuirea anumitor obiecte cu altele înaintea transmisiei. Acest lucru este folositor când se doreşte

serializarea unei clase declarate `final` care nu implementează interfaţa `Serializable`. În schimb, se poate înlocui obiectul care trebuie trimis cu un alt obiect care se poate transmite şi care ulterior este convertit înapoi la cel original. Această reconversie este rezolvată de metoda `resolveObject()` a clasei `ObjectInputStream`. Fiecare obiect care este scris într-un `Object OutputStream` va fi trimis acestei metode. Pentru a-l lăsa neschimbat, se returnează `obiect`, altfel o valoare echivalentă serializabilă.

- `protected boolean enableReplaceObject(boolean permisiune) throws SecurityException;`
  Deoarece metoda `replaceObject()` introduce anumite probleme, aceasta nu este iniţial permisă. Pentru ca înlocuirea să fie permisă este necesar întâi apelul metodei `enableReplaceObject()`; aceasta va permite (sau nu) înlocuirea obiectului conform cu parametrul `permisiune`.
- `protected void annotateClass(Class clasa) throws IOException;`
  Este utilizată de subclasele `ObjectOutputStream` pentru a scrie informaţii suplimentare la transmiterea clasei `clasa`. Metoda va apărea prima dată când un obiect din această clasă este transmis prin flux. Fluxurile de obiecte specifică, de obicei, numele clasei şi informaţii despre versiune: o subclasă poate, de exemplu, să adauge locaţia clasei codului binar Java.
- `protected void annotateProxyClass(Class clasa) throws IOException;`

Subclasele pot implementa această metodă pentru memorarea datelor obişnuite în flux împreună cu descriptorii pentru clasele *proxy* dinamice. Metoda se apelează exact o dată pentru fiecare descriptor de clasă din flux. Metoda corespunzătoare din `ObjectInputStream` este `resolveProxyClass()`.

Anumite obiecte pot refuza să fie serializate din motive de securitate şi vor arunca excepţia `NotSerializableException`, dacă se încearcă serializarea lor. În particular, când se serializează un obiect, orice câmp `private` este expus. Aceasta prezintă un risc de securitate, deoarece câmpuri `private` pot fi extrase din fluxul de transmisie a octeţilor. Din acest motiv, Java cere ca toate clasele serializabile să declare explicit acest lucru.

Mecanismul de declarare a permisiunii de serializare este implementarea interfeţei `Serializable` sau `Externalizable` (care să implementeze orice metodă cerută). Marea majoritate a nucleului claselor API suportă serializarea prin implementarea interfeţei `Serializable` (cum ar fi: `String`, `Rectangle`, `Components` etc.), dar nu şi clase cum ar fi `InputStream` sau `Socket`.

Există multe alte excepţii de tip `IOException` care pot fi aruncate de metoda `writeObject()`. Dacă apare o excepţie în timpul unui apel al lui `writeObject()`, atunci vor fi serializate nişte date speciale, astfel încât la restaurare se va primi o excepţie `WriteAbortedException` când se va încerca citirea obiectului.

5.4.2. Clasa `ObjectInputStream`

Clasa `ObjectInputStream` implementează interfața `ObjectInput` (derivată din `DataInput`) şi are toate metodele definite (în particular, `readObject()`). Ca un `FilterInputStream`, constructorul ataşează unui obiect `InputStream` existent, de unde toate datele sunt citite. Tipurile primitive sunt citite în mod normal, dar există metode noi pentru lucrul cu tipuri referință.

Clasa `ObjectInputStream` furnizează un constructor `public` şi unul `protected`:

- `ObjectInputStream(InputStream in) throws IOException;`
  Creează un obiect `ObjectInputStream` ataşat obiectului `in`. Acest constructor poate arunca o excepție `IOException`, deoarece este citit imediat un antet din fluxul ataşat pentru a asigura compatibilitatea cu obiectul de tip `Object OutputStream` utilizat la scrierea datelor.
- `protected ObjectInputStream() throws IOException, SecurityException;`
  Permite subclaselor înlocuirea protocolului de serializare implicit.

Când sunt citite tipuri primitive, pot fi utilizate metodele interfeței `DataInput`.

Exemplul 5.4.3. Iată un cod Java pentru citirea a patru octeți şi reconstituirea unui întreg:

```
ObjectInputStream in = new ObjectInputStream(
   socket.getInputStream());
int valoare;
value = in.readInt();
```

În plus față de metodele interfeței `DataInput`, clasa `ObjectInputStream` furnizează metode relative la procesul de deserializare. Acestea includ metode pentru rezolvarea fişierelor de clasă necunoscute. Metodele publice furnizate sunt toate descrise de interfața `ObjectInput`, care extinde `DataInput` adăugând şi alte metode relativ la obiecte ca şi cele din clasa `InputStream`. Iată care sunt cele mai importante metode din această clasă:

- `Object readObject() throws IOException, ClassNotFoundException;`
  Este responsabilă pentru deserializarea obiectelor dintr-un flux de comunicare.

Exemplul 5.4.4. Subprogramul Java de mai jos citeşte trei obiecte din clasele `Color`, `String` şi `Date`. Metoda `readObject()` returnează un `Object` care se converteşte la un tip convenabil. Transmiterea unui obiect `Class` în fluxul de obiecte nu înseamnă trimiterea codului binar Java al clasei respective. De fapt, se transmite doar numele clasei şi signatura (adică membrii acesteia şi câți octeți ocupă fiecare).

```
Color culoare = (Color) in.readObject;
String sir = (String) in.readObject();
Date azi = (Date) in.readObject();
```

Clasa `ObjectInputStream` furnizează protecţie împotriva încercărilor de citire greşite, cum ar fi citirea de date binare dintr-un flux utilizând metode uzuale de tip `read()` când următorul articol este un obiect sau citirea unui obiect serializat când următorul articol este o dată binară. Încercarea de citire a unei date normale, când următorul articol în flux este un obiect, va returna `EOF`. Încercarea de citire a unui obiect, când următorul articol din flux este o dată normală, va implica o excepţie `OptionalDataException`.

- `void defaultReadObject() throws IOException, ClassNotFoundException;`
  Citeşte câmpurile obiectului apelat din obiectul `ObjectInputStream`. Acesta este folosit doar de implementarea obişnuită a lui `readObject()`. Apelul acestei metode într-un mod neadecvat conduce la o excepţie `NotActiveException`.
- `protected Object resolveObject(Object ob) throws IOException;`
  Este utilizată de subclasele lui `ObjectInputStream` pentru a înlocui anumite obiecte cu altele după recepţie. Aceasta este folosită, de obicei, în asociere cu metoda `replaceObject()` a unui obiect de tip `ObjectOutputStream` pentru a permite transmisia de obiecte care nu sunt, în general, serializabile. Această metodă trebuie să returneze un obiect (`Object`) care este compatibil cu clasa originală (adică o instanţă a clasei originale sau o subclasă). De exemplu, un obiect `FileInputStream` poate fi înlocuit pentru transmisie cu un obiect `SerializedFileInputStream` care include numele fişierului asociat. După recepţie, metoda `resolveObject()` poate înlocui obiectul cu un nou `FileInputStream`.
- `protected boolean enableResolveObject(boolean permisiune) throws SecurityException;`
  Deoarece mecanismul de înlocuire implică anumite probleme de securitate relativ la accesul la datele private, acesta nu este iniţial permis. Pentru a permite rezolvarea obiectului de către un `ObjectInputStream` este necesar să apelăm metoda `enableResolveObject()`; aceasta va permite (sau nu) înlocuirea obiectului, conform valorii argumentului `permisiune`.
- `protected Class resolveClass(ObjectStreamClass osc) throws IOException, ClassNotFoundException;`
  Este folosită de subclasele lui `ObjectInputStream` pentru a citi informaţii suplimentare scrise de metoda `annotateClass()` a obiectului de tip `ObjectOutputStream` şi apoi să rezolve clasa respectivului obiect. De exemplu, un obiect `ObjectOutputStream` poate transmite codul actual al unei clase (sau locul unde se găseşte) şi un `ObjectInputStream` îl poate atunci utiliza pentru crearea clasei, dacă aceasta nu este deja disponibilă local.

În ceea ce priveşte excepţiile fluxului de obiecte, ele pornesc de la `IOException` până la accesul fluxului suport sau `ObjectStreamException` (o subclasă a lui `IOException`) relativ la procesul de serializare. Excepţiile `ObjectStreamException` sunt:

- `InvalidClassException` – indică faptul că o clasă locală nu poate fi utilizată pentru deserializarea unui obiect dintr-un flux, chiar dacă obiectul serializat aparține unei clase prezente pe calculatorul local;
- `NotSerializableException` – poate fi aruncată în timpul citirii sau scrierii şi indică faptul că obiectul care s-a citit sau s-a scris nu suportă serializarea;
- `StreamCorruptedException` – indică faptul că un flux nu conține ca date pentru flux obiecte valide sau că informația din antetul fluxului este incorectă;
- `NotActiveException` – este aruncată, dacă una dintre metodele de serializare a fluxului de obiecte obişnuit este accesată de oriunde, numai că nu de o implementare de serializare obişnuită;
- `InvalidObjectException` – este aruncată, dacă un obiect eşuează după testul de validare de după scriere;
- `OptionalDataException` – este aruncată, dacă s-a făcut o încercare de citire a unui obiect dintr-un flux care conține date primitive;
- `WriteAbortedException` – este aruncată la încercarea de citire a unui obiect care a eşuat să fie scris, pentru că a apărut o excepție în timpul procesului de scriere.

5.4.3. Interfața `Serializable`

Interfața `Serializable` este o interfață de marcare utilizată de clase pentru a indica că acestea pot fi serializate prin mecanismul implicit. Această interfață nu are date sau metode! Este doar un marcator (engl. *flag*) care informează fluxurile de obiecte că instanțele serializate ale unei clase nu vor dezvălui informații sensibile sau deschise la problemele de securitate posibile.

Dacă o subclasă a unei clase neserializabile doreşte să implementeze interfața `Serializable`, superclasa imediat următoare trebuie să furnizeze un constructor care nu are nici un argument. Este responsabilitatea subclasei să serializeze toate datele superclasei folosind următoarele metode obişnuite:

- `private void writeObject(ObjectOutputStream iesire) throws IOException;`
  Va fi apelată când o instanță a acestei clase este serializată către obiectul `iesire`. Ar trebui apelată metoda `defaultWriteObject()` pentru `iesire` înaintea scrierii datelor obişnuite proprii folosind metodele standard `ObjectOutputStream`, chiar dacă nu posedă date care vor fi serializate de mecanismul implicit. Aceasta va asigura compatibilitatea cu versiuni viitoare ale clasei care doresc să aibă avantajele opțiunii de serializare implicită;
- `private void readObject(ObjectInputStream intrare) throws IOException, ClassNotFoundException;`
  Va fi apelată când o instanță a acestei clase este deserializată din obiectul `intrare`. Ar trebui apelată metoda `defaultReadObject()` pentru `intrare` înaintea citirii datelor obişnuite proprii şi executării oricărei inițializări a câmpurilor `transient`.

Continuăm cu metode de rezoluţie şi înlocuire. Aceste metode pot fi furnizate de obiecte serializabile care doresc să conducă forma lor serializată către altă clasă. Aceasta este foarte utilă, deoarece separă detaliile de serializare de implementarea actuală a clasei. Acest lucru este important când detaliile de serializare au devenit implicate în probleme de versiuni. Aceste metode sunt:

- `private Object writeReplace() throws ObjectStreamException;` Ar trebui să returneze o copie a acestui obiect pentru scopuri de serializare. Dacă o clasă care implementează această metodă este scrisă pentru un obiect `ObjectOutputStream`, atunci această metodă va fi apelată şi va returna obiectul serializat (care trebuie să fie serializabil). De obicei, aceasta va implementa următoarea metodă `readResolve()` pentru ca obiectul să poată fi translatat înapoi către tipul original în vederea deserializării;
- `private Object readResolve() throws ObjectStreamException;` Va trebui să returneze o copie a acestui obiect în scopul deserializării. Când o clasă care implementează această metodă este citită dintr-un `ObjectInputStream`, această metodă va fi apelată şi va fi returnat obiectul rezultat. Aceasta va fi, de obicei, o instanţă a unei clase care iniţial furnizează o metodă `writeReplace()`.

5.4.4. Interfaţa **`Externalizable`**

Această interfaţă trebuie să fie implementată de clase care doresc să furnizeze facilităţi de serializare scrise de utilizator. Fluxurile de obiecte vor scrie doar numele clasei către flux; toate celelalte date trebuie comunicate prin această interfaţă. Clasele care utilizează acest mecanism nu câştigă nici o facilitate pusă la dispoziţie de mecanismul de serializare implicit.

Un obiect `Externalizable` trebuie să furnizeze un constructor fără argumente prin care o instanţă va fi creată anterior deserializării din flux. Metodele declarate în această interfaţă sunt:

- `void writeExternal(ObjectOutput iesire) throws IOException;` Va fi apelată când este serializată o instanţă a acestei clase. Aceasta poate scrie toate datele către `iesire`, folosind numeroasele metode declarate;
- `void readExternal(ObjectInput intrare) throws IOException;` Va fi apelată când este deserializată o instanţă a acestei clase. Aceasta poate citi toate datele din `intrare`, folosind numeroasele metode declarate. Crearea unei instanţe se va face de constructorul fără argumente, după care metoda va fi apelată.

Metodele de rezoluţie şi înlocuire pot fi, de asemenea, utilizate cu mecanismul de serializare `Externalizable`. O clasă `Externalizable` care doreşte să pună la dispoziţie rezoluţia şi înlocuirea poate implementa ambele metode `writeReplace()` şi `readResolve()`, după care serializarea şi deserializarea metodei corespunzătoare vor fi apelate automat.

5.4.5. Crearea unei clase serializabile

Implementarea unei clase serializabile ridică considerații de securitate, deoarece un obiect serializat își expune starea internă pentru citire și manipulare. Pot apărea și considerații de versiune, pentru că un obiect serializat poate mai târziu să fie citit înapoi într-un mediu cu o versiune mai veche sau mai nouă a fișierelor clasei actuale.

La implementarea unei clase trebuie ținut cont care părți ale stării clasei trebuie serializate. Implicit, toate datele nestatice și neocazionale (non-`transient`) vor fi serializate. Aceasta înseamnă că, dacă nu se dorește serializarea unor anumite variabile, atunci acestea trebuie declarate `transient`. Cu toate acestea, este bine să ne asigurăm în metodele `readObject()` și `writeObject()` că atributele `transient` vor fi corect inițializate după deserializare (altfel, acestea vor fi inițializate cu `0` sau `null` și ar conduce la o funcționare incorectă a programului).

Java introduce și un mecanism alternativ de declarare a câmpurilor care se doresc a fi serializate. Dacă o clasă declară un vector static din clasa `ObjectStreamField` numit `serialPersistentFields`, atunci doar câmpurile specificate în acest vector vor fi serializate. Acest mecanism se utilizează în aplicații care necesită serializare specială.

Există, în principiu, patru moduri standard diferite de creare a unei clase serializabile. Vom prezenta și varianta a cincea ca fiind cea mai generală modalitate de creare a unei clase serializabile.

5.4.5.1. *Derivarea unei clase serializabile*

Dacă derivăm o clasă care este deja serializabilă, avem prin tranzitivitate o subclasă serializabilă automat.

Exemplul 5.4.5. În programul Java de mai jos, clasa `UnApplet` este derivată din `Applet`, care la rândul ei este derivată din `Panel`, care este derivată din `Container`, ultima fiind derivată din `Component`. Cum clasa `Component` implementează interfața `Serializable`, rezultă că și clasa `UnApplet` este serializabilă. Fluxurile obiectului au grijă transparent de trimiterea tuturor stărilor appletului, inclusiv a datelor superclaselor, dar și a noilor date precum `eticheta`.

```
import java.awt.*;
import java.applet.*;

public class UnApplet extends Applet {
  private Label eticheta;

  public void init() {
    eticheta = new Label(getParameter("EtichetaUnu"));
    System.out.println(eticheta.getText());
  }
}
```

Pentru ca exemplul să funcţioneze complet, în fişierul `.html` asociat poate fi scris:

```
<applet code="UnApplet" width="300" height="300">
   <param name="EtichetaUnu" value="PrimaEticheta" />
</applet>
```

După apelarea acestui applet, în fereastra de ieşire (consolă) va fi afişat `"PrimaEticheta"`.

5.4.5.2. *Implementarea interfeţei* `Serializable`

Implementând interfaţa (de marcare) `Serializable`, noua clasă va prelua avantajele proprietăţilor de serializare a fluxurilor obiect.

Exemplul 5.4.6. Următorul program Java utilizează interfaţa `Serializable`:

```
import java.io.*;

class CercDoi implements Serializable {
  protected double x, y, raza;

  public CercDoi(double x, double y, double raza) {
    this.x = x;
    this.y = y;
    this.raza = raza;
  }

  public boolean punctInterior(double x, double y) {
    return (this.x - x) * (this.x - x) +
        (this.y - y) * (this.y - y) <= raza * raza;
  }

  public String toString() {
    return "Centrul (" + x + ", " + y + "), raza " + raza;
  }
}

public class TestCercDoiSerializare {
  public static void main(String args[]) {
    CercDoi obiect = new CercDoi(0, 0, 10);
    try {
      ObjectOutputStream oos = new ObjectOutputStream(
          new FileOutputStream("cerc.dat"));
      oos.writeObject(obiect);
      oos.close();
    }
    catch (IOException e) {
        System.err.println("Eroare la serializare");
      }
      try {
        ObjectInputStream ois = new ObjectInputStream(
          new FileInputStream("cerc.dat"));
        CercDoi obiectCitit = (CercDoi) ois.readObject();
        System.out.println(obiectCitit);
        ois.close();
      }
      catch (IOException e) {
        System.err.println("Eroare la deserializare" +
         e.getMessage());
      }
      catch (ClassNotFoundException e) {
        System.err.println("N-am gasit clasa obiectului in flux" +
        e.getMessage());
      }
    }
  }
```

Cu toate că variabilele `x`, `y`, `raza` şi `altCerc` nu sunt publice, acestea sunt accesibile fluxurilor obiect. Metoda `readObject()` returnează un `Object`, deci trebuie realizat *down-casting* prin specificarea numelui clasei obiectului citit. La execuţia programului de mai sus se va afişa:

```
Centrul (0.0, 0.0), raza 10.0
```

5.4.5.3. *Metode de serializare obişnuite*

Există multe cazuri când este util să dirijăm cumva procesul de serializare. În particular, este util când o clasă include date ce nu sunt serializabile, care sunt unice pentru o maşină virtuală şi trebuie transmise alteia sau a căror transmisie nu este necesară.

Pentru a furniza serializare de această formă, o clasă trebuie să implementeze interfaţa `Serializable` şi metodele `writeObject()` şi `readObject()`. Spre deosebire de metodele cu acelaşi nume din clasele `ObjectOutputStream`, respectiv `ObjectInputStream`, metodele definite de noi nu au aceeaşi signatură. Invers faţă de metodele din sistemul Java, acestea vor fi asociate obiectului curent al clasei care se serializează, iar ca parametru vor primi fluxul de obiect (intrare/ieşire). Acestea vor avea prototipul:

- `public void writeObject(ObjectOutputStream fluxObiectIesire) throws IOException;`
- `public void readObject(ObjectInputStream fluxObiectIntrare) throws IOException, ClassNotFoundException;`

Aceste metode trebuie declarate astfel încât să arunce orice tip de excepţie. Cu toate acestea, programatorul trebuie să se asigure că se pot arunca doar aceste tipuri de excepţii declarate de metodele corespunzătoare `writeObject()` şi `readObject()`.

De exemplu, metoda `writeObject()` poate arunca doar `IOException` sau o subclasă a acesteia, pe când `readObject()` poate arunca `IOException`, `ClassNotFoundException` sau o subclasă a acestora.

Exemplul 5.4.7. Fluxurile de obiecte nu pot transmite date `transient`, adică date care prin natura lor sunt unice. Datele `transient` (`patratulRazei` şi `id`) vor fi procesate în metodele `writeObject()` şi `readObject()`. Nici datele statice nu vor fi trimise de către fluxurile de obiecte. Datele statice sunt globale unei maşini virtuale şi deci nu are sens transmiterea acestora printr-o singură instanţă a clasei. Când se va primi fluxul de obiecte, datele statice (`globalId`) vor fi iniţializate ca şi cele normale.

```
import java.io.*;

class CercDoi implements Serializable {
  protected double x, y, raza;

  public CercDoi(double x, double y, double raza) {
    this.x = x;
    this.y = y;
    this.raza = raza;
  }

  public boolean punctInterior(double x, double y) {
    return (this.x - x) * (this.x - x) +
       (this.y - y) * (this.y - y) <= raza * raza;
  }

  public String toString() {
     return "Centrul (" + x + ", " + y + "), raza " + raza;
  }
}

class CercTrei extends CercDoi {
  private transient double patratulRazei;
  private transient int id;

  private static int globalId;
  private synchronized static int getId() {
    return globalId++;
  }

  public CercTrei() {super(0,0,0);}

  public CercTrei(double x, double y, double raza) throws
  IOException {
    super(x, y, raza);
    patratulRazei = raza * raza;
    id = getId();
  }

  public boolean punctInterior(double x, double y) {
    return (this.x - x) * (this.x - x) +
       (this.y - y) * (this.y - y) <= patratulRazei;
  }

  public void writeObject(ObjectOutputStream obiectIesire)
  throws IOException {
    obiectIesire.writeDouble(x);
    obiectIesire.writeDouble(y);
    obiectIesire.writeDouble(raza);
  }

  public void readObject(ObjectInputStream obiectIntrare)
     throws IOException, ClassNotFoundException {
    x = obiectIntrare.readDouble();
    y = obiectIntrare.readDouble();
    raza = obiectIntrare.readDouble();
    patratulRazei = raza * raza;
    id = getId();
  }

  public String toString() {
    return "Centrul (" + x + ", " + y + "), raza " + raza +
      " patratul Razei = " + patratulRazei + "\n id = " + id +
       " globalId = " + globalId;
  }
}

public class TestCercTreiSerializare {
  public static void main(String args[]) {
    CercTrei obiect = null;
    try {
      obiect = new CercTrei(1, 1, 10);
    }
    catch (IOException e) {
      System.err.println(
        "Eroare la creare obiect din Clasa CercTrei");
    }
    System.out.println("obiect = " + obiect);
    try {
        ObjectOutputStream oos = new ObjectOutputStream(
            new FileOutputStream("cerc.dat"));
        obiect.writeObject(oos);
        oos.close();
      }
      catch (IOException e) {
        System.err.println("Eroare la serializare " +
           e.getMessage());
      }
      try {
        ObjectInputStream ois = new ObjectInputStream(
           new FileInputStream("cerc.dat"));
        CercTrei obiectCitit = new CercTrei();
        obiectCitit.readObject(ois);
        System.out.println(obiectCitit);
        ois.close();
      }
      catch (IOException e) {
        System.err.println("Eroare la deserializare " +
           e.getMessage());
      }
      catch (ClassNotFoundException e) {
       System.err.println("N-am gasit clasa obiectului in flux" +
           e.getMessage());
      }
    }
  }
```

S-a marcat variabila `patratulRazei` cu `transient` pentru a face calculele din metoda `punctInterior()` mai eficient. Nu este necesară trimiterea acesteia, pentru că ea se poate reface ca fiind pătratul razei. Variabila `id` a fost declarată `transient` pentru a defini un identificator unic pentru fiecare maşină virtuală. Fiecare instanţă a clasei `CercTrei` dintr-o maşina virtuală va avea o valoare unică pentru `id`. Nu are sens să trimitem această dată, pentru că nu putem garanta că maşina virtuală primitoare nu a setat deja acest identificator.

Programul va afişa la execuţie:

```
obiect = Centrul (1.0, 1.0), raza 10.0 patratul Razei = 100.0
id = 0 globalId = 1
Centrul (1.0, 1.0), raza 10.0 patratul Razei = 100.0
id = 1 globalId = 2
```

5.4.5.4. *Implementarea interfeței* `Externalizable`

Interfața `Externalizable` este un mecanism alternativ pentru ca o clasă să furnizeze serializabilitate fără să țină cont de avantajele mecanismelor de serializare. Când un obiect `Externalizable` este trimis către un flux, se scrie un antet constând în principiu din numele clasei și atunci clasa este direct responsabilă de propria serializare și deserializare.

Exemplul 5.4.8. În acest exemplu, clasa `Dreptunghi` implementează interfața `Externalizable` și furnizează mecanisme manuale pentru serializare și deserializare. Pentru obiectele `Externalizable` trebuie să furnizăm un constructor `public` fără argumente care va crea inițial obiectul. Fluxurile obiect nu vor apela metodele unei superclase `Externalizable` (ele vor fi argumente), ci va trebui să apelăm explicit metodele superclasei asociate unui obiect din clasa `Dreptunghi` (care implementează `Externalizable`).

```
import java.io.*;

public class Extern {
  public static void main(String args[]) {
    Dreptunghi d = new Dreptunghi(5, 20);
    try {
      ObjectOutputStream oos = new ObjectOutputStream(
         new FileOutputStream("dreptunghi.dat"));
      d.writeExternal(oos);
      oos.close();
    }
    catch (IOException e) {
      System.out.println("Eroare " + e);
    }
    try {
      ObjectInputStream ois = new ObjectInputStream(
         new FileInputStream("dreptunghi.dat"));
      Dreptunghi obiectCitit = new Dreptunghi();
      obiectCitit.readExternal(ois);
      System.out.println(obiectCitit);
      ois.close();
    }
    // la citire, pot interveni mai multe tipuri de exceptii!
    catch (IOException e) {
      System.err.println("Eroare la fisierul de intrare: " + e);
    }
    catch (ExceptionInInitializerError e) {
      System.err.println("Initializarea provocata de aceasta " +
         " metoda a esuat." + " Eroare: " + e);
      }
      catch (SecurityException e) {
        System.err.println("Nu avem permisiunea sa cream o noua " +
             " instanta." + " Eroare: " + e);
      }
    }
}

class Dreptunghi implements Externalizable {
    private int latime, lungime;

    public Dreptunghi (int latime, int lungime) {
      this.latime = latime;
      this.lungime = lungime;
    }

    public Dreptunghi() {}

    public void writeExternal(ObjectOutput obiectIesire)
      throws IOException {
        obiectIesire.writeInt(latime);
        obiectIesire.writeInt(lungime);
    }

    public void readExternal(ObjectInput obiectIntrare)
      throws IOException {
        latime = obiectIntrare.readInt();
        lungime = obiectIntrare.readInt();
    }

    public String toString() {
      return "Dreptunghi de latime " + latime + " si lungime "
         + lungime;
    }
}
```

În urma execuţiei, programul va afişa:

```
Dreptunghi de latime 5 si lungime 20
```

5.4.5.5. *Declararea interfeţei şi a tuturor metodelor de acces*

Se poate „coborî" şi mai mult la nivelul serializării obiectelor şi anume declarând interfaţa de serializare care va conţine două metode abstracte:

- `public abstract void scrieDate(DataOutputStream dos);`
- `public abstract void citesteDate(DataInputStream dis);`

Se utilizează clasa `DataOutputStream`, respectiv `DataInputStream`, pentru a scrie, respectiv pentru a citi date de tipuri primitive către un, respectiv dintr-un flux. Către fluxul de ieşire se vor trimite numărul de obiecte şi numele clasei, urmate de obiectele în cauză. Acestea vor fi preluate din fluxul de intrare în aceeaşi ordine, adică numărul de obiecte, numele clasei şi apoi obiectele. Pentru crearea unui obiect de un anumit tip de clasă, apelăm metoda statică `forName()` din clasa `Class`:

```
public static Class<?> forName(String className) throws
  ClassNotFoundException;
```

sau:

```
public static Class<?> forName(String nume, boolean initializator,
   ClassLoader incarcator) throws ClassNotFoundException;
```

care returnează obiectul `Class` asociat cu nume, folosind valoarea variabilei `incarcator`. Clasa este iniţializată doar dacă valoarea lui `initializator` este `true`.

Apoi putem utiliza metoda `newInstance()` din clasa `Class` pentru a crea un obiect al unei clase fără a apela operatorul `new`. Sintaxa este:

```
public T newInstance() throws InstantiationException,
   IllegalAccessException;
```

Exemplul 5.4.9. Se vor defini 10 obiecte serializabile: primele 5 vor aparţine clasei `PrimaClasa`, iar următoarele 5 clasei `ADouaClasa`.

```
import java.io.*;

public class SerializareObiectePrimitive {
  public static void main(String args[]) {
    // declaram 10 obiecte pe care le vom serializa
    Serializare A[] = new Serializare[10];
    int i;
    // vom defini 5 obiecte din PrimaClasa si ultimele 5 din
    // ADouaClasa
    for (i = 0; i < A.length/2; i++)
      A[i] = new PrimaClasa(i, (double) i * 1.5, "test" + i,
        i * 2);
    for (i = A.length/2; i < A.length; i++)
      A[i] = new ADouaClasa(i * 3);
    /* vom declara un fisier in acces de scriere. Acesta va
       avea structura: numar de obiecte, reprezentarea
       obiectelor (numele clasei urmat de obiectul
       respectiv) */
    try {
      FileOutputStream fos = new FileOutputStream(
        "Fisier1.tmp");
  DataOutputStream dos = new DataOutputStream(fos);
  dos.writeInt(A.length);
  for (i = 0; i < A.length; i++) {
    dos.writeUTF(A[i].getClass().getName());
    A[i].scrieDate(dos);
  }
  fos.close();
}
catch (IOException e) {
  System.out.println("Eroare " + e);
}
// declaram un alt vector din interfata Serializare
Serializare B[] = null;
// vom declara un fisier in acces de citire. Acesta va fi
// citit dupa structura: numar de obiecte, reprezentarea
// obiectelor (numele clasei urmat de obiectul respectiv)
try {
  FileInputStream fis = new FileInputStream(
      "Fisier1.tmp");
  DataInputStream dis = new DataInputStream(fis);
  int numarObiecte = dis.readInt();
  B = new Serializare[numarObiecte];
  for (i = 0; i < B.length; i++) {
    Class C = Class.forName(dis.readUTF());
    B[i] = (Serializare) C.newInstance();
    B[i].citesteDate(dis);
    System.out.println(B[i]);
  }
  dis.close();
}
// la citire, pot interveni mai multe tipuri de exceptii!
catch (IOException e) {
  System.err.println("Eroare la fisierul de intrare: " + e);
}
catch (InstantiationException e) {
  System.err.println(
      "Clasa C este abstracta, interfata, sir, "
      + "tip primitiv sau void. " + "Eroare: " + e);
}
catch (IllegalAccessException e) {
  System.err.println("Clasa sau initializatorul " +
      "nu este accesibil. Eroare " + e);
}
catch (ClassNotFoundException e) {
  System.err.println("Nu am gasit clasa cu numele " +
               "respectiv. Eroare " + e);
        }
        catch (ExceptionInInitializerError e) {
          System.err.println("Initializarea provocata " +
             "de aceasta metoda a esuat. Eroare: " + e);
        }
        catch (SecurityException e) {
          System.err.println("Nu avem permisiunea sa cream " +
             "o noua instanta. Eroare: " + e);
        }
    }
}

class PrimaClasa implements Serializare {
  int intreg = 0;
  double dublu = 0.0;
  String sir = null;
  ADouaClasa obiectDoi = null;

  public PrimaClasa() {}
  public PrimaClasa(int i, double d, String s, int numarDoi) {
    intreg = i;
    dublu = d;
    sir = s;
    obiectDoi = new ADouaClasa(numarDoi);
  }

  public void scrieDate(DataOutputStream dos) {
    try {
      dos.writeInt(intreg);
      dos.writeDouble(dublu);
      dos.writeUTF(sir);
      obiectDoi.scrieDate(dos);
    }
    catch (IOException e) {
      System.err.println("Eroare: " + e);
    }
  }

  public void citesteDate(DataInputStream dis) {
    try {
      intreg = dis.readInt();
      dublu = dis.readDouble();
      sir = dis.readUTF();
      obiectDoi = new ADouaClasa();
      obiectDoi.citesteDate(dis);
    }
    catch (IOException e) {
      System.err.println("Eroare: " + e);
    }
  }

  public String toString() {
   /* in cazul lui obiectDoi se apeleaza metoda toString() din
      clasa ADouaClasa */
    return "intreg = " + intreg +  ", dublu = " + dublu +
      ", sir = " + sir + ", obiectDoi: " + obiectDoi;
  }
}

class ADouaClasa implements Serializare {
  int intreg = 0;

  public ADouaClasa() {}

  public ADouaClasa(int i) {
    intreg = i;
  }

  public void scrieDate(DataOutputStream dos) {
    try {
      dos.writeInt(intreg);
    }
    catch (IOException e) {
      System.err.println("Eroare: " + e);
    }
  }

  public void citesteDate(DataInputStream dis) {
    try {
      intreg = dis.readInt();
    }
    catch (IOException e) {
      System.err.println("Eroare: " + e);
    }
  }

  public String toString() {
    return "intreg = " + intreg;
  }
```

```
}

interface Serializare {
  public abstract void scrieDate(DataOutputStream dos);
  public abstract void citesteDate(DataInputStream dis);
}
```

Programul va afişa la execuţie:

```
intreg = 0, dublu = 0.0, sir = test0, obiectDoi: intreg = 0
intreg = 1, dublu = 1.5, sir = test1, obiectDoi: intreg = 2
intreg = 2, dublu = 3.0, sir = test2, obiectDoi: intreg = 4
intreg = 3, dublu = 4.5, sir = test3, obiectDoi: intreg = 6
intreg = 4, dublu = 6.0, sir = test4, obiectDoi: intreg = 8
intreg = 15
intreg = 18
intreg = 21
intreg = 24
intreg = 27
```

5.5. Concluzii

Capitolul de faţă se referă la fişiere de intrare/ieşire (secţiunile 5.3.1 şi 5.3.2) şi la fluxuri de intrare/ieşire (secţiunea 5.3.3). Există fluxuri de intrare/ieşire de nivel jos (subsecţiunea 5.3.3.1), fluxuri de intrare/ieşire filtru de nivel înalt (subsecţiunea 5.3.3.2) şi fluxuri de citire/scriere (subsecţiunea 5.3.3.3). Secţiunea 5.4 descrie procedeele de serializare în Java.

5.6. Test-grilă

Întrebarea 5.6.1. Care dintre afirmaţiile de mai jos sunt adevărate?

a) caracterele UTF se reprezintă pe 8 biţi;
b) caracterele UTF se reprezintă pe 16 biţi;
c) caracterele UTF se reprezintă pe 24 biţi;
d) caracterele Unicode se reprezintă pe 16 biţi;
e) caracterele Bytecode se reprezintă pe 16 biţi.

Întrebarea 5.6.2. Care dintre afirmaţiile de mai jos sunt adevărate?

a) când se creează o instanţă a lui `File` şi nu se foloseşte semantica de denumire a fişierelor de pe maşina locală, atunci constructorul va arunca o excepţie `IOException`;
b) când se creează o instanţă a lui `File` şi fişierul respectiv nu există pe sistemul de fişiere local, atunci se creează un astfel de fişier;

c) când o instanţă a lui `File` este eliberată de *garbage collector*, atunci fişierul corespunzător de pe sistemul local este şters;
d) nici una dintre afirmaţiile de mai sus nu este adevărată.

Întrebarea 5.6.3. Stabiliţi valoarea de adevăr a afirmaţiilor de mai jos:
Clasa `File` conţine o metodă care schimbă directorul de lucru curent;
Clasa `File` conţine o metodă care listează conţinutul directorului de lucru curent.

a) fals, fals;
b) adevărat, fals;
c) fals, adevărat;
d) adevărat, adevărat.

Întrebarea 5.6.4. Câţi octeţi va scrie codul Java de mai jos către fişierul `fisier.txt`?

```
try {
  FileOutputStream f1 = new FileOutputStream("fisier.txt");
  DataOutputStream f2 = new DataOutputStream(f1);
  f2.writeInt(3);
  f2.writeDouble(0.01);
  f2.close();
  f1.close();
}
```

a) 2;
b) 8;
c) 12;
d) 16;
e) numărul de octeţi va depinde de sistemul local respectiv.

Întrebarea 5.6.5. Ce va afişa subcodul Java de mai jos la linia 8?

```
1. FileOutputStream f1 = new FileOutputStream("fisier.txt");
2. for (byte b = 10; b < 50; b++) f1.write(b);
3. f1.close();
4. RandomAccessFile f2 = new RandomAccessFile("fisier.txt", "r");
5. f2.seek(10);
6. int i = f2.read();
7. f2.close();
8. System.out.println("i = " + i);
```

a) `i = 10`
b) `i = 20`
c) `i = 30`
d) se va arunca o excepţie la linia 1;
e) se va arunca o excepţie la linia 4.

Întrebarea 5.6.6. Presupunem că utilizăm codul de mai jos pentru a crea un fişier:

```
FileOutputStream f1 = new FileOutputStream("fisier.txt");
DataOutputStream f2 = new DataOutputStream(f1);
for (int i = 0; i < 500; i++)
  f2.writeInt(i);
```

Pentru a citi date din acest fişier, care dintre soluţiile de mai jos este corectă?

a) Construim un `FileInputStream` căruia i se trimite ca parametru numele fişierului. Folosind acest obiect se creează un obiect `DataInputStream` pentru care se apelează metoda `readInt();`

b) Construim un `FileReader` căruia i se trimite ca parametru numele fişierului. Folosind acest obiect se apelează metoda `readInt();`

c) Construim un `PipedInputStream` căruia i se trimite ca parametru numele fişierului. Folosind acest obiect se apelează metoda `readInt();`

d) Construim un `RandomAccessFile` căruia i se trimite ca parametru numele fişierului. Folosind acest obiect se apelează metoda `readInteger();`

e) Construim un `FileReader` căruia i se trimite ca parametru numele fişierului. Folosind acest obiect se creează un obiect `DataInputStream` pentru care se apelează metoda `readInt()`.

Întrebarea 5.6.7. Stabiliţi valoarea de adevăr a afirmaţiilor de mai jos:

Clasa `Reader` conţine o metodă care citeşte şi returnează o valoare de tip `float`;
Clasa `Writer` conţine o metodă care scrie o valoare de tip `float`.

a) fals, fals;
b) adevărat, fals;
c) fals, adevărat;
d) adevărat, adevărat.

Întrebarea 5.6.8. Fie codul Java de mai jos. Care este rezultatul execuţiei lui?

```
1. File f1 = new File("numeDirector");
2. File f2 = new File(f1, "numeFisier");
```

a) eroare de compilare la linia 2, deoarece avem un apel recursiv nepermis;

b) programul este corect, dar nu se creează nici un director şi nici un fişier;

c) programul este corect şi se creează în directorul curent un director cu numele `numeDirector`, iar în acest director se creează fişierul cu numele `numeFisier`;

d) programul este corect şi se creează în directorul curent un director cu numele `numeDirector` şi fişierul cu numele `numeFisier`;

e) programul este corect şi se creează în directorul curent doar fişierul cu numele `numeFisier`.

Întrebarea 5.6.9. Presupunem că avem un fragment de cod Java dintr-o aplicaţie care are drept de scriere în directorul de lucru curent şi că în directorul curent nu există fişierul numit `fisier.txt`. Care va fi rezultatul compilării şi execuţiei codului de mai jos?

```
1. try {
2.     RandomAccessFile f1 = new RandomAccessFile(
               "fisier.txt", "rw");
3.     BufferedOutputStream f2 = new BufferedOutputStream(f1);
4.     DataOutputStream f3 = new DataOutputStream(f2);
5.     f3.writeDouble(Math.PI);
6.     f3.close();
7.     f2.close();
8.     f1.close();
9. } catch(IOException e) { }
```

a) Codul Java de mai sus nu se poate compila;
b) Codul Java de mai sus se poate compila, dar aruncă o excepţie la linia 3;
c) Codul Java de mai sus se poate compila şi executa, dar nu are nici un efect pe sistemul de fişiere local;
d) Codul Java de mai sus se poate compila şi executa, creându-se fişierul `fisier.txt` în directorul curent.

Întrebarea 5.6.10. Presupunem că avem o clasă `C1` care extinde interfaţa `Serializable`. Selectaţi o opţiune de mai jos în funcţie de valoarea de adevăr a următoarelor afirmaţii:

1. Întotdeauna clasa `C1` trebuie să redefinească metodele `readObject()` şi `writeObject()`;
2. Dacă `C1` extinde clasa `Applet`, atunci nu este nevoie să se redefinească metodele `readObject()` şi `writeObject()`;
3. Atributele `transient` din clasa `C1` nu vor fi serializate.

a) 1, 2, 3;
b) 1, 2;
c) 1, 3;
d) 2, 3;
e) 2.

Întrebarea 5.6.11. Presupunem că avem un program Java şi că în directorul de lucru curent există drepturi de scriere. Care va fi rezultatul compilării şi execuţiei codului de mai jos?

```
import java.io.*;

class C1 {
  private double a = 3.45;
  public String toString() { return "a = " + a; }
}
public class TestGrilaSerializare {
  public static void main(String args[]) {
    C1 obiect = new C1();
    try {
        ObjectOutputStream oos = new ObjectOutputStream(
           new FileOutputStream("intrare.dat"));
        oos.writeObject(obiect);
        oos.close();
      }
      catch (IOException e) {
        System.err.println("Eroare la serializare " +
           e.getMessage());
      }
      try {
        ObjectInputStream ois = new ObjectInputStream(
           new FileInputStream("intrare.dat"));
        C1 obiectCitit = (C1) ois.readObject();
        System.out.println(obiectCitit);
        ois.close();
      }
      catch (IOException e) {
        System.err.println("Eroare la deserializare " +
           e.getMessage());
      }
      catch (ClassNotFoundException e) {
       System.err.println("N-am gasit clasa obiectului in flux" +
           e.getMessage());
      }
    }
  }
```

a) Codul Java de mai sus nu se poate compila;
b) Codul Java de mai sus se poate compila, dar aruncă o excepţie la serializare;
c) Codul Java de mai sus se poate compila, dar aruncă o excepţie la deserializare;
d) Codul Java de mai sus se poate compila, dar aruncă două excepţii (la serializare şi la deserializare);
e) Codul Java de mai sus se poate compila şi executa, afişându-se textul `a = 3.45`.

5.7. Exerciţii propuse spre implementare

Exerciţiul 5.7.1. Scrieţi un program Java care modifică programul `Gaseste.java` (exemplul 5.3.11), astfel încât să afişaţi şi numerele liniilor care conţin acel cuvânt. În final, afişaţi şi numărul total de apariţii ale cuvântului căutat. Toate aceste informaţii trebuie scrise într-un fişier extern. În cazul în care cuvântul nu este găsit în fişier, afişaţi un mesaj corespunzător.

Exerciţiul 5.7.2. Scrieţi o aplicaţie Java care listează pe ecran conţinutul unui director împreună cu precizarea faptului dacă este subdirector sau fişier, furnizând şi lungimile (în octeţi) ale fişierului respectiv. Numele directorului al cărui conţinut trebuie listat se dă ca parametru din linia de comandă. Furnizaţi numărul total de octeţi ocupaţi de director.

Exerciţiul 5.7.3. Scrieţi un program Java care citeşte un fişier text şi creează un alt fişier pornind de la primul în care liniile sunt numerotate.

Exerciţiul 5.7.4. Utilizând clasa `FileRandomAccess`, scrieţi un program Java care primeşte la intrare un fişier binar de numere întregi (sau de tip `float`, `double` etc.) şi listează fiecare element (la ecran şi în acel fişier binar) înlocuind numerele negative cu modulul acestora.

Exerciţiul 5.7.5. Scrieţi două funcţii Java care întorc `true`, dacă pointerul fişierului vizează sfârşitul fluxului, respectiv al fişierului cu acces aleator. De exemplu, acestea pot avea prototipurile:

```
public static boolean eof(InputStream fisier);
public static boolean eof(RandomAccessFile fisier);
```

Exerciţiul 5.7.6. Fie un tablou de 8 elemente de tip `int` şi altul de 12 elemente de tip `double`. Să se serializeze şi deserializeze aceşti vectori. Folosiţi toate cele 5 metode de serializare/deserializare. Care este cea mai avantajoasă?

Exerciţiul 5.7.7. Presupunem că avem un tablou de dimensiune foarte mare care nu poate fi alocat contiguu în memorie. Să se împartă în subtablouri mai mici cărora li se poate aloca spaţiu de memorie care apoi se serializează în fişiere de pe disc. Când se doreşte reutilizarea lor, acestea se deserializează de pe disc.

5.8. Proiecte propuse spre implementare

Proiectul 5.8.1. (*Pretty printing*) Scrieţi un program Java care primeşte la intrare un program Java (eventual „scris neuniform”) şi îl transformă într-un program Java scris respectând regulile stilului profesionist de redactare a programelor (indentare a subinstrucţiunilor, o singură instrucţiune pe rând, o singură linie vidă etc.).

6. Fire de execuţie

6.1. Cuvinte-cheie

- fire de execuţie, sincronizare şi paralelism
- excludere mutuală, prioritate
- clasa `Thread` şi interfaţa `Runnable`
- grupuri de fire de execuţie

6.2. Introducere

Acest capitol cuprinde câteva noţiuni introductive referitoare la programarea concurentă.

6.2.1. Breviar teoretic

Program concurent = mulţime de programe secvenţiale (task-uri, procese) care se execută într-un cadru paralel (abstract). Cadrul paralel (abstract) nu va impune ca fiecare proces să se execute pe un procesor fizic separat.

Arhitecturile de calculatoare utilizate pentru execuţia programelor concurente sunt:

- memorie comună = computere în care procesele îşi partajează accesul la locaţiile de memorie sau sunt capabile să apeleze un sistem de operare centralizat pentru a primi un serviciu. Algoritmii de tip memorie comună corespund programelor concurente care se execută prin partajarea unui singur computer sau arhitecturilor multiprocesor care partajează o memorie;
- sisteme distribuite = computere care comunică prin schimb de mesaje.

Programare concurentă abstractă = studiul execuţiei intercalate (engl. *interleaving*) a secvenţelor de instrucţiuni atomice ale mai multor procese secvenţiale. Faptul că este abstract înseamnă că avem un model idealizat al unui fenomen care ignoră detaliile instrucţiunilor-maşină specifice sau detaliile de arhitectură.

Tehnici de abstractizare utilizate în crearea sistemelor software:

- *încapsulare* = stocarea elementelor de date şi a procedurilor ce operează asupra lor într-un singur modul;
- *concurenţa* = abstractizarea necesară studiului comportării dinamice a programelor.

Secţiunea critică reprezintă o resursă a unui calculator (memorie, imprimantă etc.) la care accesul se realizează secvenţial (la un moment dat, un singur proces are acces exclusiv la respectiva resursă).

6.2.2. Corectitudinea programelor

Există două definiţii pentru corectitudinea programelor secvenţiale care se presupun că se termină. Fie P(x) proprietatea variabilelor de intrare x şi Q(x, y) proprietatea variabilelor de intrare x şi variabilelor de ieşire y.

Corectitudine parţială: (P(a) şi termină(Prog(a, b))) > Q(a, b).

Corectitudine totală: P(a) > (termină(Prog(a, b)) şi Q(a, b)).

Un program care intră întotdeauna într-o buclă infinită este parţial corect pentru orice specificare.

Există două tipuri de proprietăți de corectitudine pentru programele concurente:

1. *Safety properties* (proprietatea de siguranță) – proprietatea trebuie să fie totdeauna adevărată:
   - excludere mutuală: în secțiunea critică există (în orice moment) cel mult un proces;
   - absența deadlock-ului: dacă există N (N > = 2) procese care doresc să acceseze secțiunea critică, atunci există unul care reușește.
2. *Liveness properties* (proprietăți de continuitate) – proprietatea trebuie să fie eventual adevărată:
   - absența inaniției (engl. *starvation*): dacă un proces își anunță intenția de a intra în secțiunea critică, atunci el are șanse reale de reușită (nu trebuie să avem procese defavorizate);
   - accesarea secțiunii critice în absența disputei (engl. *contention*): dacă numai un proces dorește să acceseze secțiunea critică, atunci el trebuie să reușească.

În cazul disputelor pentru secțiunea critică, se cunosc mai multe moduri de rezolvare (proprietăți *fairness*), dintre care două pot fi implementate ușor:

- *Linear waiting*: dacă un proces face o cerere, el va fi servit înainte ca orice alt proces să fie servit de două ori;
- *FIFO waiting*: dacă un proces P face o cerere, el va fi servit înaintea oricărui proces ce face ulterior aceeași cerere (*First In, First Out*).

Dacă se cunoaște timpul maxim permis unui proces să rezolve o cerere, se va putea estima perioada de timp după care va fi servită o cerere anume. Soluția *FIFO waiting* este uzuală în sistemele centralizate.

6.2.3. Primitivele de programare concurentă

Problema excluderii mutuale pentru N procese (N = 2) se poate rezolva apelând la:

- instrucțiuni atomice (`LOAD`, `STORE` etc.). Se cunosc algoritmii Dekker, Petterson;
- primitive de programare concurentă (semafoare, monitoare).

Monitorul este primitiva de programare concurentă structurată, care concentrează răspunderea corectitudinii numai în câteva module. Etimologie: a monitoriza, a superviza.

Modul de funcționare este următorul: pentru fiecare obiect sau grup de obiecte (de exemplu, alocarea dispozitivelor de intrare/ieșire sau a memoriei, cererile de intrare/ieșire) se definește câte un monitor. Procesele cer servicii de la diferitele monitoare. Dacă același monitor este apelat de două procese, implementarea asigură că acestea sunt procesate secvențial pentru a asigura excluderea mutuală. Dacă sunt apelate mai multe monitoare, execuțiile lor se pot intercala.

Sintaxa monitoarelor se bazează pe încapsularea elementelor de date și a procedurilor care operează asupra lor într-un singur modul. Interfața cu un monitor constă într-o mulțime de proceduri care acționează asupra unor date ascunse în modul. Monitorul protejează datele interne de accese nerestricţionate şi în plus sincronizează apelurile la procedurile de interfaţă (spre deosebire de un modul obişnuit).

Pentru sincronizare definim o variabilă de condiţie C care are asociate trei operaţii:

- *Wait(C)* – procesul care cheamă procedura monitor ce conţine această instrucţiune este suspendat într-o coadă FIFO asociată cu C;
- *Signal(C)* – în cazul în care coada pentru C este nevidă, atunci activează procesul din capul cozii;
- *Nonempty(C)* – returnează adevărat când coada pentru C este nevidă.

6.3. Programare concurentă în Java

Programarea cu fire de execuţie (engl. *multithread*) este un aspect important al limbajului Java. În literatura de specialitate, sinonimele pentru *multithreading* sunt paralelism şi concurenţă.

Multitasking-ul înseamnă capacitatea unui computer de a executa mai multe programe (procese, task-uri) în acelaşi timp (de exemplu, pe sistemele de operare de tip Windows, Linux ş.a.). Multitasking-ul nu implică neapărat un sistem multiprocesor, ci poate avea loc şi pe un singur procesor. Sistemul de operare (la sistemele monoprocesor) alocă o cuantă de timp pentru execuţia unui task. Task-urile care nu se execută îşi aşteaptă rândul într-o coadă de aşteptare (create de sistemul de operare pe baza unui mecanism de priorităţi).

Multithreading înseamnă capacitatea unui program de a executa mai multe secvenţe de cod în acelaşi timp. O astfel de secvenţă de cod se numeşte fir de execuţie sau *thread*. Datorită posibilităţii creării mai multor *thread*-uri, un program Java poate să execute mai multe sarcini simultan (de exemplu, animaţia unei imagini, transmiterea unei melodii la placa de sunet, comunicarea cu un server).

Limbajul Java suportă *multithreading* prin clase disponibile în pachetul `java.lang`, care este pachetul fundamental al limbajului Java.

În pachetul `java.lang` există două clase şi o interfaţă cu care se pot dezvolta programe *multithread*: clasele `Thread` şi `ThreadGroup` şi interfaţa `Runnable`.

Clasa `Thread` şi interfaţa `Runnable` oferă suport pentru lucrul cu *thread*-uri ca entităţi separate, iar clasa `ThreadGroup`, pentru crearea unor grupuri de *thread*-uri în vederea tratării acestora într-un mod unitar.

Clasa `Thread` implementează interfaţa `Runnable`, iar obiectele de tip `ThreadGroup` conţin mai multe obiecte de tip `Thread`. Un `ThreadGroup` poate conţine mai multe `ThreadGroup`-uri.

Există două metode pentru crearea unui *thread*:

- creăm o clasă derivată din clasa `Thread`;
- creăm o clasă care implementează interfaţa `Runnable`.

6.3.1. Crearea unui fir de execuţie prin extinderea clasei **`Thread`**

Pentru ca firul de execuţie să ruleze propria metoda `run()`, trebuie să extindem clasa `Thread` şi să redefinim metoda `run()`. Există următoarele etape:

- crearea unei clase derivate din clasa `Thread`;
- suprascrierea funcţiei `public void run()` moştenită din clasa `Thread`. De exemplu, metoda `main()` este apelată de *Java Runtime System*. Metoda `run()` este metoda apelată când se execută un *thread*. Când se porneşte *Java Virtual Machine*, se porneşte un *thread* care apelează metoda `main()`. JVM rulează atât timp cât toate *thread*-urile sunt în execuţie şi nu se apelează metoda `exit()` din clasa `System`;
- instanţierea unui obiect *thread* folosind `new`;
- pornirea *thread*-ului instanţiat, prin apelul metodei `start()` moştenită din clasa `Thread`. Apelul acestei metode face ca maşina virtuală Java să creeze contextul de program necesar unui *thread* după care să apeleze metoda `run()`.

Exemplul 6.3.1. Iată un prim program demonstrativ foarte simplu:

```
public class PrimulFir {
  public static void main(String args[]) {
    System.out.println("Crearea firului de executie");
    FirExecutie fir = new FirExecutie();
    System.out.println("Startam firul de executie");
    fir.start();
    System.out.println("Revenim in main()");
  }
}

class FirExecutie extends Thread {
  public void run() {
    int numarPasi = 5;
    System.out.println("Run se executa de " + numarPasi +
      " ori.");
    for (int i = 1; i <= numarPasi; i++)
      System.out.println("Pasul " + i);
    System.out.println("Run si-a terminat treaba.");
  }
}
```

Mesajul „`Revenim in main()`" apare înaintea mesajului „`Run se executa de 5 ori`". Metoda `main()` are propriul său *thread* în care se execută. Prin apelul `start()` se cere maşinii virtuale Java crearea şi pornirea unui nou *thread*. Din funcţia `start()` se va ieşi imediat. *Thread*-ul corespunzător metodei `main()` îşi continuă execuţia independent de firul de execuţie proaspăt creat.

Exemplul 6.3.2. Putem crea mai multe fire de execuţie, chiar şi cu nume.

```
public class AlDoileaFir {
  public static void main(String args[]) {
    FirDeExecutie fir1 = new FirDeExecutie("FIR1");
    FirDeExecutie fir2 = new FirDeExecutie("FIR2");
    System.out.println(
      "JVM va crea contextul celor doua fire de executie!");
    fir1.start();
    fir2.start();
    System.out.println(
      "JVM a creat contextul celor doua fire de executie!");
  }
}

class FirDeExecutie extends Thread {
  private String numeFir;

  public FirDeExecutie(String nume) {
    numeFir = nume;
  }

  public void run() {
    int numarPasi = 5;
    System.out.println("Run se executa de " + numarPasi +
      "ori.");
    for (int i = 1; i <= numarPasi; i++) {
      System.out.println(numeFir + " este la pasul " + i);
      try {
        sleep(500);
      }
      catch (InterruptedException e) {
        System.err.println("Eroare in somn");
      }
    }
    System.out.println(numeFir + " si-a terminat treaba");
  }
}
```

Observăm că aceste fire de execuţie rulează concomitent.

6.3.2. Crearea unui *thread* utilizând interfaţa `Runnable`

Această modalitate este de foarte mare ajutor, deoarece astfel, clasa de tip `Thread` pe care o implementăm poate moşteni capabilităţi de la altă clasă (Java nu permite moştenirea multiplă). Firul de execuţie va rula metoda `run()` pentru alte obiecte. Pentru aceasta se utilizează o altă formă de constructor al clasei `Thread`:

```
public Thread(Runnable destinatie);
```

Interfaţa `Runnable` descrie o singură metodă, şi anume `run()`. Operaţiile de creare a unui *thread* folosind interfaţa `Runnable` sunt:

- crearea unei clase care să implementeze interfaţa `Runnable`; de exemplu:

```
class FirExecutie extends Afisare implements Runnable { ... }
```

- implementarea metodei `run()` din interfaţa `Runnable`;
- instanţierea unui obiect al clasei utilizând operatorul `new`:

```
FirExecutie firExecutie = new FirExecutie();
```

- crearea unui obiect din clasa `Thread` folosind un constructor care are ca parametru un obiect de tip `Runnable`:

```
Thread fir = new Thread(firExecutie);
```

- se porneşte firul de execuţie (`fir`):

```
fir.start();
```

Exemplul 6.3.3. Un exemplu simplu de utilizare a interfeţei `Runnable`:

```
/** clasa principala */
public class PrimulRunnable {
  public static void main(String args[]) {
    System.out.println("Crearea obiectului runnable");
    Tinta tinta = new Tinta();
    System.out.println("Cream firul de executie.");
    Thread fir = new Thread(tinta);
    System.out.println("Startam firul de executie.");
    fir.start();
    System.out.println("Revenim in main()");
  }
}

class Afisare {
  public void afisare(String mesaj) {
    System.out.println(mesaj);
  }
}

/** clasa corespunzatoare firelor de executie */
class Tinta extends Afisare implements Runnable {
  public void run() {
    int numarPasi = 5;
```

```
      afisare("Run executa " + numarPasi + " pasi.");
      for (int i = 1; i <= numarPasi; i++)
        afisare("Pasul " + i);
      afisare("Run si-a terminat treaba.");
    }
}
```

În urma execuţiei programului de mai sus vom obţine următorul rezultat:

```
Crearea obiectului runnable
Cream firul de executie.
Startam firul de executie.
Revenim in main()
Run executa 5 pasi.
Pasul 1
Pasul 2
Pasul 3
Pasul 4
Pasul 5
Run si-a terminat treaba.
```

Se observă faptul că imediat după pornirea firului de execuţie metoda `main()` se termină, iar firul de execuţie îşi continuă execuţia.

6.3.3. Controlul unui fir de execuţie

Un fir de execuţie se poate afla la un moment dat în una din următoarele stări:

- *new* – firul de execuţie a fost creat însă nu a fost pornit.
- *runnable* – starea de execuţie sau de aşteptare a disponibilităţii unei resurse a sistemului.
- *blocked* – stare de blocare, în care se aşteaptă accesul la o resursă partajată.
- *waiting* – starea de aşteptare la care s-a ajuns datorită apelului unei metode `wait()` sau `join()` fără a se specifica o limită de timp.
- *time waiting* – starea de aşteptare a trecerii unui anumit timp, la care s-a ajuns datorită unui apel al unei metode `sleep`, `wait()` sau `join` în care s-a specificat un interval de timp.
- *terminated* – firul de execuţie şi-a terminat execuţia.

Fiecărei stări îi corespunde un element din tipul enumerare `Thread.State`. Acestea sunt: `NEW`, `RUNNABLE`, `BLOCKED`, `WAITING`, `TIME_WAITING` şi `TERMINATED`.

Constructorii clasei `Thread` sunt:

- `Thread()`
- `Thread(Runnable target)`
- `Thread(Runnable target, String name)`
- `Thread(String name)`

- `Thread(ThreadGroup group, Runnable target)`
- `Thread(ThreadGroup group, Runnable target, String name)`
- `Thread(ThreadGroup group, Runnable target, String name, long stackSize)`
- `Thread(ThreadGroup group, String name)`

Parametrii utilizaţi au următoarea semnificaţie: `target` reprezintă un obiect care implementează interfaţa `Runnable` şi care dă partea de execuţie a firului; `name` stabileşte un nume pentru firul de execuţie; `group` indică grupul de fire execuţie din care va face parte firul curent; `stackSize` fixează dimensiunea stivei aferente firului de execuţie.

Prezentăm în tabelul de mai jos câteva dintre metodele clasei `Thread`:

Prototipul metodei	Descrierea metodei
static int **activeCount**()	Returnează numărul firelor de execuţie active din grupul firului care se execută (din cadrul căruia s-a apelat această metodă).
static Thread **currentThread**()	Returnează o referinţă la firul de execuţie care se execută.
static void **dumpStack**()	Afişează stiva cu apelurile metodelor din firul care se execută.
static int **enumerate**(Thread[] tarray)	Copie în tabloul dat ca parametru toate firele de execuţie active din grupul căruia îi aparţine firul ce se execută şi din subgrupurile acestuia.
static Map<Thread, StackTraceElement[]> **getAllStackTraces**()	Returnează tabelul asociativ dintre firele de execuţie neterminate şi stiva apelurilor metodelor (engl. *stack trace*).
long **getId**()	Obţine identificatorul firului de execuţie curent.
String **getName**()	Întoarce numele firului curent.
int **getPriotity**()	Returnează prioritatea firului curent.
StackTraceElement[] **getStackTrace**()	Returnează un tablou cu elementele din stiva apelurilor metodelor.
Thread.State **getState**()	Obţine starea curentă a firului de execuţie curent.
ThreadGroup **getThreadGroup**()	Întoarce grupul de fire de execuţie din care face parte firul curent.
static boolean **holdsLock**(Object obj)	Returnează `true` doar dacă firul care se execută deţine exclusivitatea asupra obiectului specificat.
void **interrupt**()	Întrerupe execuţia firului curent.
static boolean **interrupted**()	Verifică dacă firul care se execută a fost întrerupt.

➔

➔

boolean **isAlive**()	Returnează `true` dacă este activ (nu s-a terminat).
boolean **isDaemon**()	Verifică dacă firul curent este de tip *daemon*.
boolean **isInterrupted**()	Verifică dacă firul curent a fost întrerupt.
void **join**()	Se aşteaptă ca firul curent să se termine.
void **join**(long millis)	Se aşteaptă `millis` secunde. Pentru valoarea zero se aşteaptă oricât.
void **join**(long millis, int nanos)	Se aşteaptă cel mult `millis` milisecunde şi `nanos` nanosecunde pentru ca firul curent să se termine.
void **run**()	Dacă la apelarea constructorului s-a specificat un obiect care implementează interfaţa `Runnable`, atunci se apelează metoda `run()` a respectivului obiect. Altfel, metoda nu are nici un effect; eventual, se va redefini metoda în clasele derivate pentru stabilirea corpului de execuţie a firului de execuţie.
void **setDaemon**(boolean on)	Stabileşte dacă firul curent devine *daemon* sau nu.
void **setName**(String name)	Schimbă numele firului curent cu cel dat ca parametru.
void **setPriority**(int newPriority)	Modifică prioritatea firului de execuţie curent.
static void **sleep**(long millis)	Cedează execuţia firului current pentru `millis` milisecunde.
static void **sleep**(long millis, int nanos)	Cedează execuţia firului care se execută pentru `millis` milisecunde şi `nanos` nanosecunde.
void **start**()	Porneşte execuţia firului curent. Maşina virtuală va apela metoda `run()`.
static void **yield**()	Firul care se execută cedează execuţia.

Un fir de execuţie terminat nu mai poate fi repornit. Această încercare este privită ca o eroare de execuţie furnizând excepţia `java.lang.IllegalThreadStateException`.

Metoda `wait()` trece un fir de execuţie din starea *runnable* în starea *waiting*, iar metodele `notify()` şi `notifyAll()` scot firele de execuţie din starea *waiting*. Totuşi, aceste metode sunt foarte diferite de `join()`, `sleep()` şi `yield()`. În primul rând, pentru că sunt implementate în clasa `Object`, şi nu în clasa `Thread`. În al doilea rând, pentru că pot fi apelate doar în cod sincronizat.

Exemplul 6.3.4. Prezentăm o mică generalizare a problemei cu mingea care sare într-un dreptunghi. Programul poate starta mai multe mingi, pe care apoi le oprim (folosim un vector de cel mult 10 fire de execuţie).

```
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;

public class Bouncer extends Applet implements ActionListener {
   private static final long serialVersionUID = 1L;
   private Button start, stop;
   private Ball firExecutie[] = new Ball[10];
   private static int numarFirExecutie = 0;

   /** Metoda de initializare a appletului.
    * Creeaza butoanele interfetei cu utilizatorul.
    */
   public void init() {
      start = new Button("Start (cel mult 10 mingi)");
      add(start);
      start.addActionListener(this);
      stop = new Button("Stop");
      add(stop);
      stop.addActionListener(this);
   }

   public void actionPerformed(ActionEvent event) {
      if (event.getSource() == start)
         if (numarFirExecutie < 10) {
            Graphics g = getGraphics();
            firExecutie[numarFirExecutie] = new Ball(g);
            firExecutie[numarFirExecutie].start();
            numarFirExecutie ++;
         }
      if (event.getSource() == stop)
         if (numarFirExecutie > 0)
            firExecutie[--numarFirExecutie].pleaseStop();
   }
}

class Ball extends Thread {
   /** Indica daca firul de executie
    * trebuie sa ruleze in continuare.*/
   private boolean keepGoing;

   /** Contextul grafic utilizat la desenare*/
   private Graphics g;

   /** Coordonata pe axa Ox a mingii*/
```

```
private int x = 37;
/** Pasul de miscare pentru axa Ox*/
private int xChange = 7;
/** Coordonata pe axa Ox a mingii*/
private int y = 32;
/** Pasul de miscare pentru axa Ox*/
private int yChange = 2;

/** Diametrul mingii*/
private int diameter = 10;
/* Coordonatele chenarului in care se misca mingea */
private int rectLeftX = 30, rectRightX = 130;
private int rectTopY = 30, rectBottomY = 130;

/** Constructor fara parametri */
public Ball(Graphics graphics) {
  g = graphics;
  keepGoing = true;
}

/** Metoda pentru oprirea unei mingi*/
public void pleaseStop() {
  // indicam firului de executie sa se opreasca
  keepGoing = false;
  // stergem mingea
  g.setColor(Color.white);
  g.fillOval(x, y, diameter, diameter);
}

/** Realizeaza actiunea de miscare a mingii.*/
public void run() {
  // desenam chenarul
  g.drawRect(rectLeftX, rectTopY, rectRightX - rectLeftX + 10,
      rectBottomY - rectTopY + 10);
  while (keepGoing) {
    // stergem mingea
    g.setColor(Color.white);
    g.fillOval(x, y, diameter, diameter);

    // calculam noile coordonate
    if (x + xChange <= rectLeftX)   xChange =  - xChange;
    if (x + xChange >= rectRightX)  xChange =  - xChange;
    if (y + yChange <= rectTopY)    yChange =  - yChange;
    if (y + yChange >= rectBottomY) yChange =  - yChange;
    x += xChange;
```

```
        y += yChange;

        // desenam mingea
        g.setColor(Color.red);
        g.fillOval(x, y, diameter, diameter);

        // oprim firul de executie pentru 50 milisecunde
        try {
          Thread.sleep(50);
        }
        catch (InterruptedException e) {
          System.err.println("Sleep exception");
        }
      }
    }
  }
```

6.3.4. Prioritatea firelor de execuţie

Fiecare fir de execuţie are o prioritate de execuţie. În general, firul cu prioritatea cea mai mare este cel care va accesa primul resursele sistemului. Clasa `Thread` oferă trei constante de tip `int` pentru priorităţi:

- `MAX_PRIORITY` – prioritatea maximă pentru un fir de execuţie.
- `MIN_PRIORITY` – prioritatea cea mai mică pentru un fir de execuţie.
- `NORM_PRIORITY` – prioritatea normală, care este şi cea implicită.

Mai avem două funcţii importante pentru obţinerea, respectiv stabilirea priorităţii: `getPriority()` şi `setPriority()`. Pentru mai multe detalii se poate consulta tabelul din secţiunea anterioară.

Exemplul 6.3.5. Acest exemplu evidenţiază modalitatea de lucru cu priorităţile firelor de execuţie:

```
public class Prioritati {
  public static void main(String args[]) {
    // crearea a trei fire de executie
    Fir fir1 = new Fir("FIR1");
    Fir fir2 = new Fir("FIR2");
    Fir fir3 = new Fir("FIR3");
    // stabilirea prioritatilor pentru firele de executie
    fir1.setPriority(Thread.MIN_PRIORITY);
    fir2.setPriority(Thread.MAX_PRIORITY);
    fir3.setPriority(9);
    System.out.println("Firele de executie au fost create!");
```

```
      // pornirea firelor de executie
      fir1.start();
      fir2.start();
      fir3.start();
      System.out.println("Firele de executie au fost pornite!");
    }
}

// clasa definita de programator pentru fire de executie
class Fir extends Thread {
  // constructorul clasei
  public Fir(String nume) {
    super(nume); // apelul metodei superclasei
  }

  public void run() {
    int numarPasi = 5;
    // preia numele firului de executie
    String numeFir = getName();
    for (int i = 1; i <= numarPasi; i++) {
      System.out.println(numeFir + " este la pasul " + i);
      try {
        // oprirea pentru 0,5 secunde a firului de executie
        sleep(500);
      }
      catch (InterruptedException e) {
        System.err.println("Eroare la somn...");
      }
    }
    System.out.println(numeFir + " si-a terminat treaba");
  }
}
```

Execuţia programului anterior va conduce la afişarea următorului rezultat:

```
Firele de executie au fost create!
FIR2 este la pasul 1
FIR3 este la pasul 1
Firele de executie au fost pornite!
FIR1 este la pasul 1
FIR2 este la pasul 2
FIR3 este la pasul 2
FIR1 este la pasul 2
FIR2 este la pasul 3
FIR3 este la pasul 3
FIR1 este la pasul 3
```

```
FIR2 este la pasul 4
FIR3 este la pasul 4
FIR1 este la pasul 4
FIR2 este la pasul 5
FIR3 este la pasul 5
FIR1 este la pasul 5
FIR2 si-a terminat treaba
FIR3 si-a terminat treaba
FIR1 si-a terminat treaba
```

Remarcăm faptul că la orice pas ordinea de execuție a firelor de execuție este dată de prioritatea aleasă. Se observă că `FIR2` este cel mai prioritar, iar `FIR1` este cel mai puțin prioritar.

Exemplul 6.3.6. Priorități bazate pe funcția `yield()`; primul fir de execuție cedează, uneori, procesorul pentru celelalte fire de execuție:

```
import static java.lang.System.*;

public class Renuntare {
  public static void main(String args[]) {
    // crearea firelor de executie
    FirNormal fir1 = new FirNormal("FIR1");
    FirNormal fir2 = new FirNormal("FIR2");
    FirNormal fir3 = new FirNormal("FIR3");

    // pornim firele de executie
    fir1.start();
    fir2.start();
    fir3.start();
  }
}

class FirNormal extends Thread {
  public FirNormal(String nume) {
    super(nume);
  }

  public void run() {
    String numeFir = getName();
    for (int i = 1; i <= 5; i++) {
      out.println(numeFir + " este la pasul " + i);
      // cercetam daca este primul fir
      if (numeFir.equals("FIR1")) {
        // sansa va lua valorile 0 sau 1
        int sansa = (int) (Math.random() * 2);
```

```
          out.println("sansa = " + sansa);
          if (sansa == 1) yield();
         // uneori cedam procesorul altor fire de executie (2, 3)
        }
        try {
          sleep(500);
        }
        catch (InterruptedException e) {
          err.println("Eroare in somn...");
        }
      }
      out.println(numeFir + " si-a terminat treaba");
    }
}
```

Un rezultat posibil al execuţiei programului de mai sus este:

```
FIR1 este la pasul 1
FIR3 este la pasul 1
FIR2 este la pasul 1
sansa = 1
FIR3 este la pasul 2
FIR2 este la pasul 2
FIR1 este la pasul 2
sansa = 0
FIR3 este la pasul 3
FIR2 este la pasul 3
FIR1 este la pasul 3
sansa = 1
FIR3 este la pasul 4
FIR2 este la pasul 4
FIR1 este la pasul 4
sansa = 0
FIR2 este la pasul 5
FIR3 este la pasul 5
FIR1 este la pasul 5
sansa = 0
FIR2 si-a terminat treaba
FIR3 si-a terminat treaba
FIR1 si-a terminat treaba
```

Atunci când `sansa` ia valoarea `1`, primul fir va ceda procesorul în favoarea celorlalte două fire. Acest lucru iese în evidenţă la primii doi paşi din rezultatul obţinut anterior.

În general, la metodele care conţin operaţiuni de intrare/ieşire sunt posibile blocări.

Exemplul 6.3.7. Fie următorul cod Java:

```
try {
   /** crearea unei conexiuni cu un server */
   Socket sock = new Socket("numeServer", 5000);
   /** obtinerea unui flux de intrare cu ajutorul caruia
      citim datele trimise de server */
   InputStream is = sock.getInputStream();
   /** citirea unui octet*/
   int b = is.read();
}
catch (IOException e) { ... }
```

Instrucțiunea `int b = is.read();` încearcă să citească un octet dintr-un flux de intrare conectat la portul `5000` al mașinii numite `numeServer`. Dacă octetul este disponibil, atunci el va fi citit și execuția programului continuă. Dacă nu, atunci apelul lui `read()` trebuie să aștepte. În acest caz, firul de execuție va intra în coada de așteptare, cedând execuția.

6.3.5. Grupurile de fire de execuție

Conceptul de grup de fire de execuție este asemănător cu conceptul de director de fișiere. Grupurile de *thread*-uri sunt utile pentru că oferă:

- posibilitatea tratării unitare a mai multor fire care fac parte din grup;
- sprijin în implementarea mecanismului de securitate.

Un grup de fire de execuție poate conține și alte grupuri de fire de execuție, creând un sistem ierarhic. La rădăcina acestei ierarhii se află un grup de fire de execuție numit grup de *thread*-uri sistem.

Când o aplicație Java este pornită, JVM creează un grup inițial de *thread*-uri numit grup de *thread*-uri sistem (grup sistem). În acest grup, JVM creează un alt grup de *thread*-uri numit grup *main*, în care apoi creează un *thread* (numit `main`) a cărui funcție `run()` apelează metoda `main()` a aplicației. După aceea, aplicația poate trece la crearea propriilor sale *thread*-uri. Implicit, aceste fire de execuție fac parte din grupul de *thread*-uri `main`.

Java implementează conceptul de grup de fire de execuție utilizând clasa `ThreadGroup`. Ca și *thread*-urile, un `ThreadGroup` are priorități, poate fi de tip *daemon* și are metode de control al firelor de execuție din grup. Există în plus o prioritate maximă. Un fir de execuție care face parte din acest grup nu poate avea nivel de prioritate mai mare decât acest maxim.

Constructorii clasei `ThreadGroup` sunt:

- `ThreadGroup(String name)`
- `ThreadGroup(ThreadGroup parent, String name)`

unde `name` reprezintă numele grupului, iar `parent` este grupul-părinte.

Exemplul 6.3.8. Crearea unui grup de fire de execuţie:

```
public class GrupThread {

  public static void main(String args[]) {
    ThreadGroup toate = new ThreadGroup("toate firele");
    ThreadGroup jumaGrup = new ThreadGroup(toate, "Juma de fire");
    ThreadGroup jumiGrup = new ThreadGroup(toate, "Jumi de fire");

    FirDinGrup fir1 = new FirDinGrup(toate, "FIR1");
    FirDinGrup fir2 = new FirDinGrup(toate, "FIR2");
    FirDinGrup fir3 = new FirDinGrup(jumaGrup, "FIR3");
    FirDinGrup fir4 = new FirDinGrup(jumaGrup, "FIR4");
    FirDinGrup fir5 = new FirDinGrup(jumiGrup, "FIR5");
    FirDinGrup fir6 = new FirDinGrup(jumiGrup, "FIR6");
    System.out.println(
      "Pornim firele de executie!");
    fir1.start();
    fir2.start();
    fir3.start();
    fir4.start();
    fir5.start();
    fir6.start();

    // dorim sa exemplificam cautarea unui thread intr-un grup
    // de thread-uri, sa zicem cu numele "FIR5"
    int numarFire = toate.activeCount();
    Thread fire[] = new Thread[numarFire];
    toate.enumerate(fire, true); // intreaga ierarhie
    int i;
    boolean ok = false; // ok = false <=> firul a fost gasit
    for (i = 0; i < numarFire; i++)
      if (fire[i].getName().equals("FIR5")) {
        ok = true;
        break;
    }
    if (ok) {
      System.out.print("Am gasit FIR5 pe pozitia " + i);
      System.out.println(" in cadrul grupului.");
    }
    else
      System.out.println("Nu am gasit FIR5");
  }
}
```

```
class FirDinGrup extends Thread {
  public FirDinGrup(ThreadGroup grup, String nume) {
    super(grup, nume);
  }

  public void run() {
    System.out.println(getName() + " si-a inceput executia.");
    for (int i = 1; i < 3; i++) {
      System.out.println(getName() + " este la pasul " + i);
      try {
        sleep(500);
      }
      catch (InterruptedException e) {
        System.err.println("Eroare in somn...");
      }
    }
    System.out.println(getName() + " si-a terminat treaba");
  }
}
```

Din exemplul de mai sus se poate vedea faptul că firele de execuţie sunt rulate „simultan". De exemplu, se poate întâmpla ca între cele două mesaje („`Am gasit FIR5 pe pozitia 4`" şi, respectiv „`in cadrul grupului.`"), care sunt afişate în urma execuţiei unor instrucţiuni consecutive din cadrul aceluiaşi fir, să se interpună alte mesaje date de alte fire de execuţie (de exemplu, „`FIR5 este la pasul 1`" etc.).

6.3.6. Excluderea mutuală şi sincronizarea

Presupunem că două fire de execuţie incrementează valoarea unui întreg partajat (engl. *shared*). Pot apărea probleme în sensul că ambele fire incrementează în acelaşi timp întregul, deci se pot sări valori ale acestuia. Problema se rezolvă dacă, la un moment dat, cel mult un fir are acces la acea dată. Această cerinţă se numeşte *excludere mutuală*.

Modul în care Java rezolvă excluderea mutuală constă în specificarea metodelor care partajează variabile ca fiind `synchronized` în antetul lor. Aceste metode trebuie să fie din aceeaşi clasă. Orice fir de execuţie, care încearcă să acceseze variabila partajată într-o metodă `synchronized` a unui obiect atât timp cât metoda este folosită, este blocat până când primul fir părăseşte metoda.

Codul complet al rezolvării acestei probleme este realizat de clasele `Contor1`, `Contor2`, `NumarPartajat`, `DoiContori` din exemplul de mai jos. Clasa `DoiContori` este derivată din clasa `Applet`. În clasa `NumarPartajat`, am definit metoda care poate fi accesată de cel mult un fir de execuţie:

```
public synchronized void increment()
```

Exemplul 6.3.9. Fişierul `DoiContori.java` este:

```
import java.applet.Applet;
import java.awt.Button;
import java.awt.Graphics;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class DoiContori extends Applet implements ActionListener {
  private Button start;
  private int contor = 0;

  public void init() {
    start = new Button("Start");
    add(start);
    start.addActionListener(this);
  }

  public void actionPerformed(ActionEvent event) {
    if (event.getSource() == start) {
      contor++;
      Graphics g = getGraphics();
      NumarPartajat contor12 = new NumarPartajat(g, contor);
      Contor1 contor1 = new Contor1(contor12);
      Contor2 contor2 = new Contor2(contor12);
      contor1.start();
      contor2.start();
    }
  }
}
```

Fişierul `Contor1.java` este:

```
public class Contor1 extends Thread {
  private NumarPartajat contor1;

  public Contor1(NumarPartajat contor1) {
    this.contor1 = contor1;
  }

  public void run() {
    for (int i = 1; i <= 10; i++)
      contor1.increment();
  }
}
```

Fişierul `Contor2.java` este:

```
public class Contor2 extends Thread {
  private NumarPartajat contor2;

  public Contor2(NumarPartajat contor2) {
    this.contor2 = contor2;
  }

  public void run() {
    for (int i = 1; i <= 10; i++)
      contor2.increment();
  }
}
```

Fişierul `NumarPartajat.java` este:

```
import java.awt.Graphics;

public class NumarPartajat {
  private int n = 0;
  private Graphics g;
  private int x = 0;
  private int contor;

  public NumarPartajat(Graphics g, int contor) {
    this.g = g;
    this.contor = contor;
  }

  public synchronized void increment() {
    n = n + 1;
    g.drawString(n + ", ", x * 20, 30 + 15 * contor);
    x++;
  }
}
```

6.3.7. Metodele `wait()` şi `notify()`

Tot o problemă de excludere mutuală este exemplul producător-consumator. Producătorul furnizează date pe care consumatorul le utilizează mai departe. Ce se întâmplă dacă vitezele de producere şi de consum diferă? Prin `synchronized` am rezolvat problema accesului simultan la o dată (excluderea mutuală). Dacă producătorul este mai rapid, el va citi o dată înscrisă de primul de mai multe ori.

Rezolvarea se realizează cu metodele `wait()`, `notify()` şi `notifyAll()` ale clasei `Object`. Apelul lui `wait()` va trece obiectul apelat în starea *blocked*. El rămâne blocat până la apelul unei metode `notify()` sau `notifyAll()` pentru acelaşi obiect. Condiţia necesară pentru a se apela una dintre aceste metode este ca apelul lor să se facă în interiorul metodelor `synchronized`.

Există metodele `wait()` prin care se poate specifica o durată de timp maximă de aşteptare. În acest caz, firul rămâne blocat până când timpul expiră sau alt fir apelează `notify()`.

Exemplul 6.3.10. Dorim să scriem pe ecran două texte, fiecare corespunzător unui fir de execuţie. Metoda `askFor()` din clasa `ScreenController` joacă rol de monitor (Hoare, 1974). (Monitorul este un obiect special, care permite numai unui singur fir să apeleze o metodă a lui, la un moment dat.)

Fişierul `Initial.java` este:

```
import java.applet.Applet;
import java.awt.Graphics;

public class Initial extends Applet {
  public void init() {
    Graphics g = getGraphics();
    ScreenController screen = new ScreenController();

    Text1 text1 = new Text1(g, screen);
    Text2 text2 = new Text2(g, screen);
    text1.start();
    text2.start();
  }
}
```

Fişierul `ScreenController.java` este:

```
public class ScreenController {
  private boolean inUse = false;

  public synchronized void askFor() {
    while (inUse)
      try {
        wait();
      }
      catch(InterruptedException e) {
        System.err.println("Exceptie!");
      }
    inUse = true;
  }

  public synchronized void relinquish() {
    inUse = false;
```

```
      notify();
    }
}
```

Fişierul `Text1.java` este:

```
import java.awt.Graphics;

class Text1 extends Thread {
  private Graphics g;
  private ScreenController screen;

  public Text1(Graphics g, ScreenController screen) {
    this.screen = screen;
    this.g = g;
  }

  public void run() {
    while (true) {
      screen.askFor();
      draw();
      screen.relinquish();
    }
  }

  private void draw() {
    g.drawString("Acesta este primul document", 10, 100);
    g.drawString("pe care dorim sa-l afisam la ecran", 10, 115);
  }
}
```

Fişierul `Text2.java` este:

```
import java.awt.Graphics;

class Text2 extends Thread {
  private Graphics g;
  private ScreenController screen;

  public Text2(Graphics g, ScreenController screen) {
    this.screen = screen;
    this.g = g;
  }

  public void run() {
    while (true) {
      screen.askFor();
```

```
        draw();
        screen.relinquish();
      }
    }

    private void draw() {
    g.drawString("Acesta este al doilea document", 10, 25);
    g.drawString("pe care dorim sa-l afisam apoi la ecran", 10, 40);
    }
  }
```

Exemplul 6.3.11. Tot în categoria producător-consumator intră și o aplicație ceas (digital) în care am definit două fire de execuție: *Secunde* și *Minute*. Firul Secunde este cel care se află aproape tot timpul în secțiunea critică. După 59 de secunde, se apelează `notify()` și firul Secunde părăsește secțiunea critică. Firul Minute, care „bate la ușa" secțiunii critice, intră acum și incrementează minutele. Codul complet este:

Fișierul `Ceas.java` este:

```
import java.awt.*;
import java.applet.Applet;

public class Ceas extends Applet {
  public void init() {
    Graphics g = getGraphics();
    TicTac minuteTic = new TicTac();
    Minute minute = new Minute(g, minuteTic);
    minute.start();
    Secunde secunde = new Secunde(g, minuteTic);
    secunde.start();
  }
}
```

Fișierul `Secunde.java` este:

```
import java.awt.*;

public class Secunde extends Thread {
  private Graphics g;
  private int secunde = 0;
  private TicTac minuteTic;

  public Secunde(Graphics g, TicTac minuteTic) {
    this.g = g;
    this.minuteTic = minuteTic;
  }
```

```
    public void run() {
      while (true) {
        try {
          Thread.sleep(1000);
        }
        catch(InterruptedException e) {
          System.err.println("Exceptie !");
        }
        if (secunde == 59) {
          minuteTic.tic();
          secunde = 0;
        }
        else secunde++;
        g.clearRect(10, 0, 100, 20);
        g.drawString(secunde + " secunde", 10, 20);
      }
    }
  }
```

Fişierul `Minute.java` are liniile:

```
import java.awt.*;

public class Minute extends Thread {
  private Graphics g;
  private int minute = 0;
  private TicTac minuteTic;

  public Minute(Graphics g, TicTac minuteTic) {
    this.g = g;
    this.minuteTic = minuteTic;
  }

  public void run() {
    while (true) {
      minuteTic.waitForTic();
      if (minute == 59)
        minute = 0;
      else
        minute++;
      g.clearRect(10, 20, 100, 20);
      g.drawString(minute + " minute", 10, 40);
    }
  }
}
```

Fişierul `TicTac.java` este:

```
public class TicTac {
  private boolean ticHappens = false;

  public synchronized void waitForTic() {
    while (!ticHappens)
      try {
        wait();
      }
      catch (InterruptedException e) {
        System.err.println("Exceptie la TicTac !");
      }
    ticHappens = false;
  }

  public synchronized void tic() {
    ticHappens = true;
    notify();
  }
}
```

Exemplul 6.3.12. Un program Java pentru simularea unei cafenele. Informaţiile sunt trimise de la un *thread* la altul. O comandă se realizează prin apăsarea unui buton (*burger*, *fries*, *cola*). Comanda se afişează pe ecran şi apoi se introduce într-o coadă care se afişează pe ecran. Bucătarul (*chef*) ia câte o comandă din coadă în sistemul primul venit, primul servit. Chelnerul acceptă o comandă, o afişează şi o inserează în coadă.

Fişierul `Cafe.java` este:

```
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;

public class Cafe extends Applet implements ActionListener {
  private Button burger, fries, cola, cooked;
  private Order order, complete;

  public void init() {
    Graphics g = getGraphics();
    burger = new Button("Burger");
    add(burger);
    burger.addActionListener(this);
    fries = new Button("Fries");
    add(fries);
    fries.addActionListener(this);
    cola = new Button("Cola");
```

```
      add(cola);
      cola.addActionListener(this);
      cooked = new Button("Cooked");
      add(cooked);
      cooked.addActionListener(this);
      order = new Order();
      Queue queue = new Queue(g);
      complete = new Order();
      Waiter waiter = new Waiter(g, order, queue);
      waiter.start();
      Chef chef = new Chef(g, complete, queue);
      chef.start();
   }

   public void actionPerformed(ActionEvent event) {
      if (event.getSource() == burger)
         order.notifyEvent("burger");
      if (event.getSource() == fries)
         order.notifyEvent("fries");
      if (event.getSource() == cola)
         order.notifyEvent("cola");
      if (event.getSource() == cooked)
         complete.notifyEvent("cooked");
   }
}
```

Fişierul `Waiter.java` este:

```
import java.awt.*;

public class Waiter extends Thread {
   private Order order;
   private Graphics g;
   private Queue queue;

   public Waiter(Graphics g, Order order, Queue queue) {
      this.order = order;
      this.g = g;
      this.queue = queue;
   }

   public void run() {
      g.drawString("O noua comanda", 10, 50);
      while (true) {
         String nouaComanda = order.waitForEvent();
         g.clearRect(10, 50, 50, 25);
```

```
        g.drawString(nouaComanda, 10, 70);
        try {
          Thread.sleep(5000);
        }
        catch (InterruptedException e) {
          System.err.println("Exceptie Waiter !");
        }
        if (!queue.isFull())
          queue.enter(nouaComanda);
        queue.enter(nouaComanda);
      }
    }
}
```

Fişierul `Order.java` este:

```
public class Order {
  private String order = "";

  public synchronized void notifyEvent(String nouaComanda) {
    order = nouaComanda;
    notify();
  }

  public synchronized String waitForEvent() {
    while (order.equals(""))
      try {
        wait();
      }
      catch (InterruptedException e) {
        System.err.println("Exceptie Order!");
      }
    String nouaComanda = order;
    order = "";
    return nouaComanda;
  }
}
```

Fişierul `Queue.java` este:

```
import java.awt.*;

public class Queue {
  private Graphics g;
  private String[] queue = new String[20];
  private int count = 0;
```

```
   public Queue(Graphics g) {
      this.g = g;
   }

   public synchronized void enter(String item) {
      queue[count] = item;
      count++;
      display();
      notify();
   }

   public synchronized String remove() {
      while (count == 0)
         try {
            wait();
          }
         catch (InterruptedException e) {
            System.err.println("Exceptie Queue!");
          }
      String item = queue[0];
      count --;
      for (int c = 0; c < count; c++)
         queue[c] = queue[c + 1];
      display();
      return item;
   }

   public synchronized boolean isFull() {
      return count == queue.length;
   }

   private void display() {
      g.drawString("Coada", 120, 50);
      g.clearRect(120, 50, 50, 220);
      for (int c = 0; c < count; c++)
         g.drawString(queue[c], 120, 70 + c * 20);
   }
}
```

Fişierul `Chef.java` este:

```
import java.awt.*;

public class Chef extends Thread {
   private Graphics g;
   private Order complete;
```

```
  private Queue queue;

  public Chef(Graphics g, Order complete, Queue queue) {
    this.g = g;
    this.complete = complete;
    this.queue = queue;
  }

  public void run() {
    g.drawString("Coocking", 200, 50);
    while (true) {
      String order = queue.remove();
      g.clearRect(200, 55, 50, 25);
      g.drawString(order, 200, 70);
      String cookedInfo = complete.waitForEvent();
      g.clearRect(200, 55, 50, 25);
    }
  }
}
```

6.4. Studii de caz: problema filosofilor şi problema producător-consumator

6.4.1. Problema filosofilor

Cinci filosofi stau la o masă rotundă pe care se află corespunzător cinci farfurii şi doar cinci furculiţe (de fapt, beţe chinezeşti). Filosofii pot face două lucruri: să mănânce sau să mediteze. Un filosof poate mânca din farfuria din faţa sa dacă poate lua cele două beţişoare, din stânga şi din dreapta. Descrieţi un algoritm care să rezolve problema astfel încât:

- doi filosofi să nu poată deţine acelaşi beţişor simultan;
- să nu apară deadlock;
- nici un filosof să nu moară de foame;
- dacă există filosofi care suferă (temporar) de anorexie, ceilalţi să nu fie afectaţi.

Iată o soluţie posibilă, folosind monitoare, în pseudocod:

```
// definitia monitorului
monitor MonitorFurculite {
  // furculita[i] = numarul de furculite de care dispune
  // filosoful i

  furculita : array (0 .. 4) of Integer range 0 .. 2 = (2, 2,
    2, 2, 2)
```

```
   // Condition este variabila de conditie
   OK_mananca : array (0 .. 4) of Condition;

   procedure IaFurculite(i : Integer) {
     if furculita(i) <> 2 then Wait(OK_mananca(i));
     furculita((i + 1) mod 5) := furculita((i + 1) mod 5) - 1;
     furculita((i - 1) mod 5) := furculita((i - 1) mod 5) - 1;
   }

   procedure LasaFurculite(i : Integer) {
     furculita((i + 1) mod 5) := furculita((i + 1) mod 5) + 1;
     furculita((i - 1) mod 5) := furculita((i - 1) mod 5) + 1;
     if furculita((i + 1) mod 5) = 2 then Signal(OK_mananca
       ((I + 1)mod 5));
     if furculita((i - 1) mod 5) = 2 then Signal(OK_mananca
       ((I - 1)mod 5));
   }
} // sfarsit definitie MonitorFurculite

// definitia proceselor filosof
proces Filosof(i) {
  while (true) {
    gandeste;
    IaFurculite(i);
    mananca;
    LasaFurculite(i);
  }
}
```

Codul-sursă Java este dat în continuare. Fişierul-sursă `CinciFilosofi.java` este:

```
import java.applet.Applet;
import java.awt.*;

// definitia clasei CinciFilosofi extinsa din clasa Applet
public class CinciFilosofi extends Applet {
  public void init() {
    // setam un font vizibil
    Font f = new Font("Courier", Font.BOLD, 50);
    setFont(f);
    // setam dimensiunea appletului
    setSize(400, 350);
    // preluam in variabila g contextul grafic curent
    Graphics g = getGraphics();
    // declaram si definim obiectul masa ce reprezinta
    // sectiunea critica
    Furculite masa = new Furculite();
    // declaram si definim un vector de fire de executie
    // corespunzator celor 5 filosofi
    Thread filosof[] = new Thread[5];
    int i; // variabila de lucru
    for (i = 0; i < filosof.length; i++) {
      // primul argument al constructorului este numele
      // firului de executie curent;
      // al doilea argument este contextul grafic;
      // al treilea este masa (sectiunea critica);
      // al patrulea si al cincilea argument reprezinta
      // cele doua furculite. Pentru al cincilea filosof
      // este necesară operaţia modulo 5 (pentru furculita a
      // doua).
      filosof[i] = new Filosof("Filosoful " + (i + 1), g,
         masa, i, (i + 1) % filosof.length);
    }
    // startam cele 5 fire de executie care vor lucra
    // in paralel
    for (i = 0; i < filosof.length; i++)
      filosof[i].start();
  } // sfarsit public void init()
} // sfarsit public class CinciFilosofi extends Applet

class Filosof extends Thread {
  // g reprezinta contextul grafic
  private Graphics g;
  // masa reprezinta sectiunea critica
  private Furculite masa;
  // fUnu si fDoi reprezinta cele doua furculite
  private int fUnu, fDoi;

  // constructorul clasei Filosof
  public Filosof(String nume, Graphics g, Furculite masa,
     int fUnu, int fDoi) {
    // se apeleaza constructorul Thread(nume) din clasa
    // de baza Thread. Numele Thread-ului se poate acum
    // obtine cu getName().
    super(nume);
    // folosind referinta this se definesc valorile datelor
    // private din clasa Filosof
    this.g = g;
    this.masa = masa;
    this.fUnu = fUnu;
```

```
      this.fDoi = fDoi;
   } // sfarsitul definitiei constructorului clasei Filosof

   // definitia metodei run() ce trebuie suprascrisa. run()
   // este metoda abstracta in interfata Runnable. Clasa
   // Thread o implementeaza.
   public void run() {
      // definim numeFilosof prin apelul metodei getName()
      // din clasa Thread
      String numeFilosof = getName();
      // obtinem ultimul caracter din numele Thread-ului.
      // Acesta este de fapt numarul filosofului.
      char numarCharFilosof = numeFilosof.charAt(10);
      // convertim un char la un int
      int numarFilosof = (int) numarCharFilosof - '0';
      // presupunem ca filosofii mananca sau gandesc pentru
      // totdeauna
      while (true) {
         // facem apel la metoda cereFurculitele din clasa
         // Furculite
         masa.cereFurculitele(fUnu, fDoi);
         // stergem bucata de ecran pe care vom afisa apoi mesajul
         // ca filosoful curent mananca!
         g.clearRect(15, 60 * (numarFilosof - 1) + 10, 800, 70);
         g.drawString(numeFilosof + " mananca!", 10, 60 *
            numarFilosof);
         // il lasam 100 milisecunde sa manance
         try {
            Thread.sleep(100);
         }
         catch (InterruptedException e) {
            System.err.println("Exceptie de tip Sleep: " +
               numeFilosof);
         }
         // dupa ce "s-a saturat" de mancat, apelam metoda
         // elibereazaFurculitele() din clasa Furculite
         masa.elibereazaFurculitele(fUnu, fDoi);
         // stergem bucata de ecran pe care vom afisa apoi
         // mesajul ca filosoful curent se gandeste!
         g.clearRect(15, 60 * (numarFilosof - 1) + 10, 800, 70);
         g.drawString(numeFilosof + " gandeste!", 10, 60 *
         numarFilosof);
      }
   } // sfarsitul definitiei metodei run()
} // sfarsitul definitiei clasei Filosof
```

Fişierul `Furculite.java` este:

```
// definitia clasei Furculite ce implementeaza sectiunea
// critica a problemei filosofilor
public class Furculite {
  // declaram un vector de 5 furculite
  private boolean[] furculita = new boolean[5];

  // constructorul clasei
  public Furculite() {
    // la inceput, toate furculitele sunt "pe masa".
    // De aceea, le initializam cu false. Astfel,
    // furculita[i] = true <=> furculita i este
    // ridicata de un filosof.
    for (int i = 0; i < 5; i++)
      furculita[i] = false;
  }

  // metoda cereFurculitele de parametri fUnu si fDoi.
  // Aceasta este etichetata synchronized astfel incat
  // cel mult un Thread o executa la un moment dat.
 public synchronized void cereFurculitele(int fUnu, int fDoi) {
    // atata timp cat furculita "din stanga" sau cea
    // "din dreapta" sunt ridicate de alt filosof,
    // asteapta la coada.
    while ((furculita[fUnu]) || (furculita[fDoi]))
      try {
       // asteptarea se face prin apelul lui wait() din clasa
       // Object care va trece obiectul apelat in starea Blocked
        wait();
      }
      catch(InterruptedException e) {
       System.err.println("Exceptie de la cerut furculitele !");
      }
     // dupa ce cele doua furculite au fost puse pe masa,
     // filosoful curent le ridica.
    furculita[fUnu] = true;
    furculita[fDoi] = true;
  } // sfarsitul definitiei metodei cereFurculitele()

  // metoda elibereazaFurculitele de parametri fUnu si fDoi.
  // Aceasta este etichetata synchronized astfel incat
  // cel mult un Thread o executa la un moment dat.
 public synchronized void elibereazaFurculitele(int fUnu, int
 fDoi) {
    // filosoful curent a terminat de mancat si lasa pe masa
```

```
        // cele doua furculite
        furculita[fUnu] = false;
        furculita[fDoi] = false;
        // apelam metoda notify() din clasa Object pentru a scoate
        // primul obiect din coada de asteptare Ready
        notify();
    } // sfarsitul definitiei metodei elibereazaFurculitele()
} // sfarsitul definitiei clasei Furculite
```

6.4.2. Problema producător-consumator

În anumite tipuri de aplicaţii se poate întâmpla ca o porţiune de program să producă valori care apoi sunt consumate de o porţiune diferită a aceluiaşi program. Un astfel de aranjament se numeşte relaţie producător-consumator. Java furnizează un mod elegant de organizare a acestui tip de program prin folosirea firelor de execuţie multiple şi a *pipe*-urilor. Producătorul şi consumatorul rulează fiecare propriul fir de execuţie, comunicând printr-un *pipe*.

Un *pipe* este o zonă de date de tip buffer folosită atât pentru citiri, cât şi pentru scrieri. Un *pipe* poate memora doar un număr limitat de valori. Atât citirea, cât şi scrierea într-un *pipe* implică suspendarea temporară a unui fir de execuţie. O scriere va fi suspendată, dacă bufferul *pipe* curent este plin, în timp ce o operaţie de citire va fi suspendată, dacă bufferul este vid. În ambele cazuri, execuţia va continua când condiţia este rezolvată (când cititorul eliberează spaţiu din buffer, respectiv când scriitorul adaugă un element în buffer).

În Java, *pipe*-urile sunt reprezentate în clasele `PipedInputStream`, respectiv `PipedOutputStream`. Fiecare *pipe* se manifestă printr-o pereche de pointeri la fluxuri. Valoarea a doua creată (*pipe*-ul de intrare sau ieşire) are ca argument prima valoare şi astfel se creează o conexiune între două *pipe*-uri:

```
PipedInputStream intrare = new PipedInputStream();
PipedOutputStream iesire = new PipedOutputStream(in);
```

Valorile pot fi scrise secvenţial într-un flux *pipe* de ieşire, ca şi în alte tipuri de fluxuri de ieşire. Aceste valori pot fi citite din fluxul *pipe* de intrare în aceeaşi ordine în care au fost inserate.

Problema pe care o discutăm este cea a generării numerelor mai mici decât un număr precizat (de exemplu, 1 000 000) care sunt prime şi numere Fibonacci (în acelaşi timp). Amintim că:

- un număr este prim dacă nu are alţi divizori în afară de 1 şi el însuşi;
- un număr Fibonacci este dat de recurenţa liniară de ordin 2:
  $f_0 = 0$; $f_1 = 1$; (uneori se alege $f_0 = 1$!)
  $f_n = f_{n-2} + f_{n-1}$;

Pentru fiecare dintre aceste secvenţe de numere este creat câte un fir de execuţie. Un *thread*, de fapt, nu are nevoie să ştie că secvenţa de numere are de-a face cu un *pipe*. Firul *GenereazaFibonacci* creează o secvenţă de valori din şirul lui Fibonacci pe care apoi le scrie într-un flux de ieşire. Folosind un `DataOutputStream`, putem scrie uşor valori întregi. Firul `GenereazaPrim` creează o secvenţă de valori numere prime între 2 şi un număr maxim precizat a priori. Programul principal (`PipeTest`) va crea şi va porni cele două fire de execuţie în metodele:

```
private DataInputStream apelFibonacci(int maximNumere)
```

şi:

```
private DataInputStream apelPrim(int maximNumere)
```

Acestea vor face şi legătura dintre fluxurile de intrare/ieşire în *pipe*-ul respectiv (după cum am discutat mai sus).

Constructorul clasei `PipeTest` va citi din *pipe*-uri câte un întreg natural reprezentând numărul Fibonacci curent sau numărul prim curent. Dacă sunt egale, atunci se va afişa un mesaj precum că numărul citit este Fibonacci şi prim în acelaşi timp. Dacă numărul Fibonacci citit din *pipe*-ul Fibonacci este mai mic decât numărul prim citit din *pipe*-ul numerelor prime, atunci se va citi doar din *pipe*-ul Fibonacci (în caz contrar, din *pipe*-ul numerelor prime). Execuţia programului se încheie când numărul Fibonacci curent este mai mic decât un număr maxim curent. Se preferă comparaţia cu Fibonacci (şi nu cu cele prime), deoarece acestea sunt mai rare (adică diferenţa dintre două numere Fibonacci este mai mare decât diferenţa dintre două numere prime consecutive).

Observaţie. Două rezultate din teoria numerelor:

Se cunoaşte teorema de distribuţie a numerelor prime. Notând cu p_n al n-lea număr prim, se ştie că limita p_n/n * ln n este 1, pentru n tinzând la infinit. Cât despre şirul lui Fibonacci, lucrurile sunt mai clare. Se cunoaşte chiar formula exactă a termenului general. Notând cu f_n al n-lea număr Fibonacci, avem f_n = (1/sqrt(5))* * [((1 + sqrt(5))/2)^{n + 1} – ((1 – sqrt(5))/2)^{n + 1}]. Atât p_n, cât şi f_n sunt strict crescătoare. În plus, se observă că şirul f_n este mai rapid crescător decât p_n (deoarece lim p_n/f_n = 0).

Codul-sursă Java este:

```
import java.io.*;

// definim clasa GenereazaFibonacci derivata din Thread
// aceasta are drept scop afisarea si transmiterea catre Pipe
// a urmatorului numar Fibonacci
class GenereazaFibonacci extends Thread
{
  private DataOutputStream iesire;
  private int maximNumere;

  public GenereazaFibonacci(DataOutputStream iesire, int
  maximNumere) {
      this.iesire = iesire;
      this.maximNumere = maximNumere;
   }

   public void run() {
      // initializam primele doua valori din sirul
      // lui Fibonacci
      int n = 0;
      int m = 1;
      try {
         iesire.writeInt(m);
         int nouaValoare;
         while (m < maximNumere) {
            nouaValoare = n + m;
            n = m;
            m = nouaValoare;
            System.out.println("Noul numar Fibonacci = " +
               nouaValoare);
            iesire.writeInt(nouaValoare);
         }
         iesire.close();
      }
      catch (IOException e) {
         return;
      }
   } // de la metoda public void run()
} // sfarsitul clasei GenereazaFibonacci

// definim clasa GenereazaPrim derivata din Thread
// aceasta are drept scop afisarea si transmiterea catre Pipe
// a urmatorului numar prim
class GenereazaPrim extends Thread {
   private DataOutputStream iesire;
   private int maximNumere;

   public GenereazaPrim(DataOutputStream iesire, int
   maximNumere) {
      this.iesire = iesire;
      this.maximNumere = maximNumere;
   }

   public void run() {
      // initializam nouaValoare cu 1
      int nouaValoare = 1;
      boolean estePrim; // estePrim = true <=> numarul curent
```

```
                          // este prim
      int i; // variabila de lucru
      try {
        while (nouaValoare < maximNumere) {
          nouaValoare ++;
          estePrim = true;
          for (i = 2; i * i <= nouaValoare; i++)
            if (nouaValoare % i == 0) {
              estePrim = false;
              break;
            }
          if (estePrim) {
           System.out.println("Noul numar prim = " + nouaValoare);
            iesire.writeInt(nouaValoare);
          }
        }
        iesire.close();
      }
      catch (IOException e) {
        return;
      }
    } // de la metoda public void run()
  } // sfarsitul clasei GenereazaPrim

  // definim clasa PipeTest care va folosi un pipe pentru
  // comunicarea dintre cele doua fire de executie
  public class PipeTest {
    static public void main(String [] args) {
      BufferedReader in;
      String linie;
      int maximNumere = 0;
      try {
        in = new BufferedReader(new InputStreamReader(
          System.in), 1);
        System.out.flush();
        System.out.println("Dati numarul maxim de numere: ");
        linie = in.readLine();
        maximNumere = Integer.parseInt(linie);
        in.close();
      }
      catch (IOException e) {
        System.err.println("Citire gresita de la tastatura " +
          e.toString());
      }
      PipeTest obiect = new PipeTest(System.out, maximNumere);
```

```
} // sfarsitul metodei static public void main(String [] args)

private PipeTest(PrintStream iesire, int maximNumere) {
   DataInputStream fibonacci = apelFibonacci(maximNumere);
   DataInputStream prim = apelPrim(maximNumere);
   try {
     int x = fibonacci.readInt();
     int y = prim.readInt();
     while (x < maximNumere) {
       if (x == y) {
         iesire.println("numarul " + x +
           " este fibonacci si prim !!");
         x = fibonacci.readInt();
         y = prim.readInt();
       }
       else
         if (x < y)
           x = fibonacci.readInt();
         else
           y = prim.readInt();
     }
   }
   catch (IOException e) {
     System.exit(0);
   }
} // sfarsitul definitiei metodei private PipeTest()

private DataInputStream apelFibonacci(int maximNumere) {
   try {
     // cream un generator de numere Fibonacci
     PipedInputStream intrare = new PipedInputStream();
     PipedOutputStream iesire = new PipedOutputStream(
        intrare);
     Thread firFibonacci = new GenereazaFibonacci(
        new DataOutputStream(iesire), maximNumere);
     firFibonacci.start();
     return new DataInputStream(intrare);
   }
   catch (IOException e) {
     return null;
   }
} // sfarsitul metodei private apelFibonacci()

private DataInputStream apelPrim(int maximNumere) {
   try {
        // cream un generator de numere prime
        PipedInputStream intrare = new PipedInputStream();
        PipedOutputStream iesire = new PipedOutputStream
           (intrare);
        Thread firPrim = new GenereazaPrim(
           new DataOutputStream(iesire), maximNumere);
        firPrim.start();
        return new DataInputStream(intrare);
      }
      catch (IOException e) {
        return null;
      }
    } // sfarsit metoda private DataInputStream apelPrim()
} // sfarsitul clasei public class PipeTest
```

6.5. Concluzii

Firele de execuţie permit executarea simultană a mai multor secvenţe de cod. Firele de execuţie pot partaja aceleaşi resurse, iar accesul la acestea poate fi concurent sau secvenţial. Pentru a nu apărea probleme de consistenţă, este bine ca modificarea datelor comune să se realizeze secvenţial. Acest lucru este posibil datorită sistemului de sincronizare pus la dispoziţie de mediul Java.

Sistemul pe care sunt executate programele *multithread* nu trebuie să fie neapărat multiprocesor. Sistemele de operare existente se ocupă de gestiunea proceselor şi a firelor de execuţie.

În funcţie de importanţa operaţiilor efectuate, un fir de execuţie poate fi mai prioritar decât altele la primirea resurselor necesare rulării de la sistemul de operare.

Mai multe fire de execuţie pot fi grupate pentru a fi tratate unitar. Un grup poate conţine la rândul său şi alte grupuri de fire de execuţie.

6.6. Test-grilă

Întrebarea 6.6.1. Care dintre afirmaţiile următoare sunt false?

a) Un fir de execuţie nu poate crea un alt fir de execuţie.
b) O aplicaţie poate crea cel mult 10 fire de execuţie.
c) Două fire de execuţie nu pot citi simultan date din acelaşi fişier.
d) Firele de execuţie nu pot avea obiecte comune.
e) Pentru fiecare fir de execuţie trebuie să definim o clasă.

Întrebarea 6.6.2. Cuvântul-cheie `syncronized` se referă la:

a) sincronizarea comunicării între firele de execuţie;
b) specificarea datelor care vor fi accesibile tuturor firelor de execuţie;

c) excluderea mutuală;
d) proprietatea de continuitate.

Întrebarea 6.6.3. Sincronizarea firelor de execuţie se realizează cu ajutorul:
a) cuvântului-cheie `syncronized`;
b) datelor membre statice;
c) metodelor `wait()`, `notify()` şi `notifyAll()`;
d) metodelor `sleep()`, `notify()` şi `notifyAll()`.

Întrebarea 6.6.4. Un fir de execuţie poate intra în starea de blocare astfel:
a) prin apelul metodei `sleep()`;
b) automat, de către sistemul de operare;
c) prin apelul metodei `block()`;
d) prin apelul metodei `wait()`.

6.7. Exerciţii propuse spre implementare

Exerciţiul 6.7.1. Scrieţi un program Java (care foloseşte fire de execuţie) pentru simularea unui cronometru. Acesta are două butoane, unul pentru start şi continuarea secvenţei de timp şi un al doilea pentru revenirea acului indicator la poziţia iniţială.

Exerciţiul 6.7.2. Scrieţi un program Java care generează două fire de execuţie pentru parcurgerea unui `String` de la cele două capete. Folosiţi doi pointeri (`ptrStg` şi `ptrDr`) a căror valoare se incrementează, respectiv decrementează, într-o funcţie din memoria comună (declarată într-o funcţie `synchronized`). `String`-ul se află în memoria comună. Firele de execuţie se întâlnesc la „mijloc".

Exerciţiul 6.7.3. Scrieţi un applet Java care generează la apăsarea unui buton o minge colorată (distinctă de precedentele) care se „izbeşte" şi „ricoşează" de marginile unui dreptunghi. Se vor genera cel mult N mingi (N, de exemplu, 20). Apăsând un alt buton, acestea se pot opri din mişcarea lor (în ordinea inversă generării lor). Eventual, se va permite ca mingile să se ciocnească.

Exerciţiul 6.7.4. Scrieţi un applet Java care poate genera trei mingi colorate cu roşu, galben şi albastru atunci când se apasă câte un buton (eventual denumite *roşu*, *galben* şi *albastru*). Aceste mingi se „izbesc" şi „ricoşează" într-un dreptunghi dat. Dacă se reapasă un buton precedent, atunci mingea generată anterior se opreşte, apărând în locul ei alta de aceeaşi culoare. Furnizaţi câte un buton *Stop* pentru fiecare dintre aceste trei culori, menit să oprească mişcarea mingii de culoarea respectivă. Puneţi la dispoziţie şi câte o bară de defilare necesară pentru modificarea vitezei mingilor colorate.

6.8. Proiecte propuse spre implementare

Proiectul 6.8.1. (*Problema filosofilor*) Rescrieţi programul de la problema filosofilor (vezi secţiunea 5.4.1.) în cazul în care beţişoarele sunt în centrul mesei.

Proiectul 6.8.2. (*Breakout*) Scrieţi un program Java, care foloseşte fire de execuţie, pentru simularea jocului Breakout. Acesta constă dintr-un suport care se mişcă orizontal, având la început o bilă situată pe el. Deasupra suportului (la o anumită distanţă mai sus) există un zid din cărămizi care urmează a fi doborât de către bilă. În mod obişnuit, bila poate sparge doar o cărămidă. Conform cu poziţia şi unghiul unde bila loveşte suportul, aceasta va ricoşa înapoi către zid sub un unghi egal cu unghiul de cădere, dar în sens suplementar (invers). Dacă nu se reuşeşte „prinderea" bilei de către suport, atunci acea bilă îşi încetează mişcarea pe ecran. Cărămizile pot fi obişnuite, pot conţine bonusuri (de exemplu, dacă reuşiţi să prindeţi bila care a atins cărămida bonificată, atunci bila va sparge întreaga lăţime a zidului) sau pot genera mai multe bile.

7. Appleturi şi instrumente de lucru cu ferestrele (AWT)

7.1. Cuvinte-cheie

- appleturi, navigator, pagină Web
- AWT (*Abstract Window Toolkit*)
- componente grafice şi evenimente
- gestionari de poziţionare, ferestre
- manager de securitate

7.2. Appleturi

Am văzut în capitolul 1 că appleturile sunt încărcate de pe un calculator aflat la distanţă (server) şi apoi executate local (în navigatorul Web). Din punctul de vedere al securităţii, appleturile sunt foarte restrictive relativ la următoarele operaţii:

- nu pot rula nici un program executabil local;
- nu pot citi sau scrie în sistemul de fişiere al calculatorului local;
- pot comunica doar cu serverul originar;
- pot afla doar o mulţime restrictivă de informaţii despre calculatorul local. De exemplu, pot determina numele şi versiunea sistemului de operare, dar nu şi numele utilizatorului sau adresa acestuia de e-mail.

7.2.1. Clasa `Applet` (`java.applet.Applet`)

Un applet este un mic program Java care este menit pentru inserare în alte aplicaţii. Clasa `Applet` trebuie să fie superclasa oricărui applet care va fi înglobat în paginile Web (sau rulat de `appletviewer`). Această clasă face parte din pachetul `java.applet` şi extinde clasa `Panel` (care la rândul ei are ca superclase `Container`, `Component` şi `Object`).

Un applet este introdus în cadrul unei pagini Web prin intermediul marcatorului `<applet>`. Acesta poate avea următoarele atribute:

Nume atribut	Valoare atribut
`arhive`	Enumerare de arhive separate prin virgulă.
`code`	Conţine numele clasei appletului.
`codebase`	URL-ul directorului în care se află fişierul `.class` corespunzător appletului.
`object`	Indică un fişier care conţine un applet serializat.
`alt`	Specifică un text care va fi afişat de navigatoarele care nu suportă appleturi.
`name`	Stabileşte un nume pentru applet.
`align`	Indică modalitatea de aliniere (este similar cu atributul `align` de la marcatorul `<img>` şi poate avea valorile: `top`, `center`, `bottom`).
`width`	Stabileşte lăţimea appletului în pixeli.
`height`	Stabileşte înălţimea appletului în pixeli.
`vspace`	Indică distanţa care este lăsată înainte pe verticală şi după applet.
`hspace`	Indică distanţa care este lăsată pe orizontală în stânga şi în dreapta appletului.

Exemplul 7.2.1. Un applet poate fi inserat într-o pagină Web astfel:

```
<applet code="TestApplet.class"
   width="300" height="400"
   vspace="30" hspace="50"
   alt="Un applet pentru test." align="top">
</applet>
```

Astfel, appletul va avea 300 pixeli lungime şi 400 înălţime, se va lăsa o margine de 30 pixeli înainte şi 30 după, 50 la stânga şi 50 la dreapta. Dacă marcatorul va fi succedat de un text, acesta va apărea lângă applet în partea de sus. Dacă navigatorul nu suportă appleturi, atunci se va afişa textul stabilit de atributul `alt`.

Cele mai importante metode din clasa `Applet` sunt următoarele:

Prototipul metodei	Descrierea metodei
`public void` **`init`**`()`	Este prima metodă apelată de navigator sau de `appletviewer`. Apoi se apelează metoda `start()`. Există navigatoare care apelează acestă metodă de mai multe ori.
`public void` **`start`**`()`	Se apelează imediat după metoda `init()` şi la reluarea execuţiei appletului (de exemplu, când se revine în pagină).
`public void` **`stop`**`()`	Se apelează la suspendarea activităţii appletului. Unele navigatoare nu apelează această metodă.
`public void` **`destroy`**`()`	Se apelează la terminarea execuţiei appletului.

Implicit, apelul acestor metode nu are efect. Pentru a asocia anumite acţiuni pentru applet, va trebui să redefinim metodele prezentate în tabelul anterior. De asemenea, pentru a putea desena pe suprafaţa appletului, va trebui să redefinim şi metoda `paint()`, care este moştenită din clasa `Container`. Metoda are un singur parametru de tip `java.awt.Graphics`, care se mai numeşte şi *contextul grafic*. Ori de câte ori se doreşte desenarea appletului, se va apela metoda `paint()`. Aceasta este apelată de navigator sau de utilitarul `appletviewer`. Mai multe informaţii privitoare la desenare se găsesc în subcapitolul 7.3.

Exemplul 7.2.2. Următorul applet evidenţiază momentele când sunt apelate metodele discutate anterior. Este bine să încercaţi acest exemplu cu ajutorul unui navigator şi să probaţi operaţiile de minimizare, maximizare.

```
import java.applet.Applet;
import java.awt.Graphics;

public class TestApplet extends Applet {
  String buf = "";
  public void init() {
    buf+= "init(); ";
    System.out.println("Appletul s-a initializat!");
  }
  public void paint(Graphics g) {
    g.drawString(buf,10,15);
    buf+= "  ";
    System.out.println("Appletul s-a desenat!");
  }
  public void start() {
    buf+= "start(); ";
    System.out.println("Appletul a pornit!");
  }
  public void stop() {
    buf+= "stop();";
    System.out.println("Appletul s-a oprit!");
  }
  public void destroy() {
    System.out.println("Appletul s-a terminat!");
  }
}
```

Afişarea în suprafaţa de desenare a appletului se realizează doar atunci când se apelează metoda `paint()`. Apelurile `println()` afişează mesajele la consolă (pentru a vedea consola trebuie să umblăm la opţiunile de setare a navigatorului, iar dacă avem mediul JDK instalat, va trebui să apară în *System tray bar* în Windows).

Clasa `Applet` conţine câteva metode utile, cum ar fi:

Prototipul metodei	Descrierea metodei
`public String` **`getAppletInfo`**`()`	Returnează informaţii despre applet. Implicit returnează `null`, însă programatorul poate defini metoda pentru a furniza numele autorului, termenii de copyright şi versiunea appletului.
`public URL` **`getCodeBase`**`()`	Returnează URL-ul directorului în care se găseşte appletul.
`public URL` **`getDocumentBase`**`()`	Returnează URL-ul documentului HTML în care este înglobat appletul.
`public boolean` **`isActive`**`()`	Indică dacă appletul este activ. Apelul metodei `start()` face ca appletul să fie activ, iar metoda `stop()` inactivează appletul.
`public void` **`resize`**`(int lungime, int inaltime)`	Redimensionează suprafaţa de afiliere a appletului.
`public void` **`showStatus`**`(String mesaj)`	Afişează mesajul specificat în bara de stare a navigatorului Web.

7.2.2. Obţinerea parametrilor de intrare ai appleturilor

Java pune la dispoziţie citirea parametrilor pentru un applet din fişierul XHTML asociat. În acest fişier, în interiorul tagului `<applet>` se pot scrie linii de forma:

```
<param name="numeSir" value="valoareSir" />
```

Exemplul 7.2.3. Specificarea parametrilor unui applet în fişierul HTML se poate realiza astfel:

```
<applet class="Something.class">
  <param name="arg1" value="test" />
  <param name="arg2" value="100" />
</applet>
```

Aceşti parametri se pot obţine în applet prin apelarea metodei `getParameter()` care are un argument de tip `String`, ce trebuie să fie numele unui parametru (cel care este specificat de atributul `name`) şi întoarce valoarea acestuia (dată de atributul `value` corespunzător), care este tot de tip `String`.

Valoarea parametrului `arg1` din exemplul 7.2.3 poate fi obţinută astfel:

```
String valoareArg1 = getParameter("arg1");
```

Se va obţine şirul `test`.

Dacă în fişierul HTML nu găsim nici un parametru cu numele respectiv, atunci `getParameter()` întoarce valoarea `null`.

Exemplul 7.2.4. Testarea, dacă un parametru a fost specificat, se poate realiza astfel:

```
s = getParameter("arg1");
if (s == null) s = "Valoare implicita";
```

sau

```
if ((s = getParameter("arg1")) == null)
  s = "Valoare implicita";
```

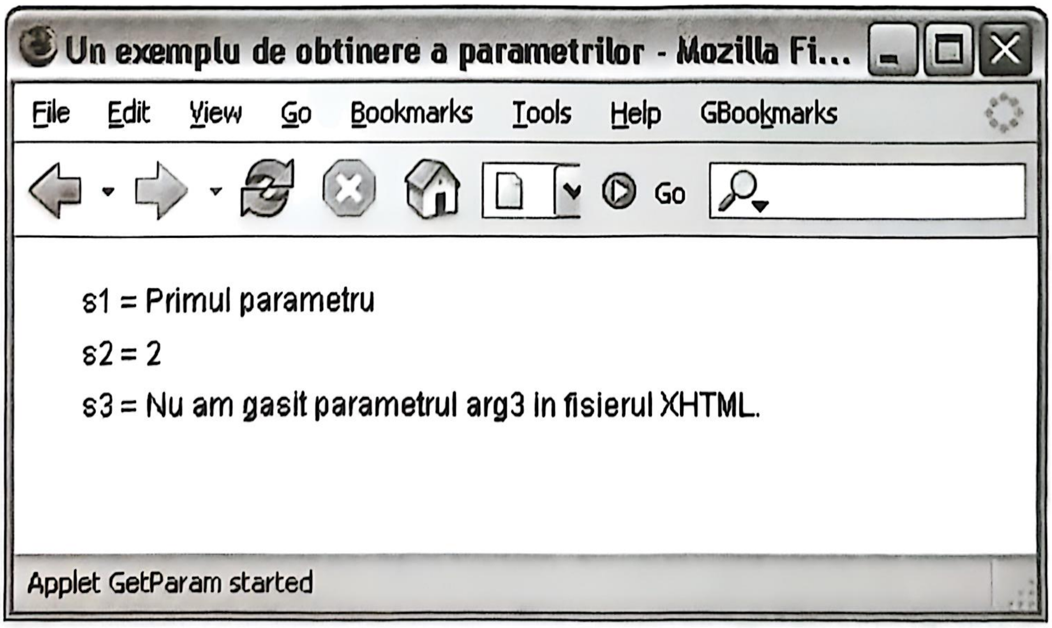
Exemplul 7.2.5. Prezentăm un applet care afişează valoarea a trei parametri: `arg1`, `arg2` şi `arg3`. Fişierul XHTML are următorul conţinut:

```
<html>
<head>
  <title>Un exemplu de obtinere a parametrilor</title>
</head>

<body>
  <applet code="GetParam.class" width="400" height="100">
    <param name="arg1" value="Primul parametru">
    <param name="arg2" value="2">
  </applet>
</body>
</html>
```

Fişierul Java corespunzător appletului este următorul:

```
import java.applet.*;
import java.awt.*;

public class GetParam extends Applet {
  String s1, s2, s3;

  public void init() {
    if ((s1 = getParameter("arg1")) == null)
      s1 = "Nu am gasit parametrul arg1 in fisierul XHTML.";
    if ((s2 = getParameter("arg2")) == null)
      s2 = "Nu am gasit parametrul arg2 in fisierul XHTML.";
    if ((s3 = getParameter("arg3")) == null)
      s3 = "Nu am gasit parametrul arg3 in fisierul XHTML.";
  }

  public void paint(Graphics g) {
    g.drawString("s1 = " + s1, 20, 20);
    g.drawString("s2 = " + s2, 20, 40);
    g.drawString("s3 = " + s3, 20, 60);
  }
}
```

În cadrul navigatorului Web, appletul va apărea astfel:


Un exemplu de obtinere a parametrilor - Mozilla Fi...
File Edit View Go Bookmarks Tools Help GBookmarks
Go

```
s1 = Primul parametru
s2 = 2
s3 = Nu am gasit parametrul arg3 in fisierul XHTML.
```

Applet GetParam started

Obţinerea parametrilor în applet

7.2.3. Contextul unui applet

Contextul unui applet este dat de interfaţa `AppletContext` din pachetul `java.applet`. Aceasta conţine toate appleturile din acelaşi document, iar metodele sale sunt utile pentru aflarea informaţiilor referitoare la mediul în care se execută appletul.

Metoda `getAppletContext()` din clasa `Applet` returnează o referinţă la un obiect care implementează interfaţa `AppletContext`. Astfel, având un applet, îi putem determina cu uşurinţă contextul. Cele mai importante metode pe care le oferă interfaţa `AppletContext` sunt:

Prototipul metodei	Descrierea metodei
`public Applet` **`getApplet`**`(String name)`	Returnează o referinţă la appletul pentru care s-a stabilit numele specificat (valoare dată de atributul `name`). Dacă nu există appletul căutat se va întoarce `null`.
`public Enumeration<Applet>` **`getApplets`**`()`	Returnează o enumerare a tuturor appleturilor din pagina Web curentă.
`public void` **`showDocument`**`(URL url)`	Se va încărca în fereastra navigatorului documentul de la URL-ul specificat. Dacă appletul nu este rulat de un navigator, atunci apelul acestei metode este ignorat.
`public void` **`showDocument`**`(URL url, String target)`	În funcţie de ţinta specificată, documentul de la URL-ul precizat se va încărca în cadrul (frame-ul) în care se află appletul (`target="_self"`), în cadrul părinte (`target="_parent"`), în ferestra navigatorului (`target="_top"`), într-o ferestră nouă fără nume (`target="_blank"`) sau într-o ferestră cu numele dat de `target` (dacă are o valoare diferită de cele de mai sus).
`public void` **`showStatus`**`(String mesaj)`	Afişează mesajul specificat în bara de stare a navigatorului.

Exemplul 7.2.6. La încărcarea paginii `FII_site.html`, appletul va redirecta navigatorul către situl Facultăţii de Informatică din Iaşi. Conţinutul paginii Web este:

```
<html>
<head>
  <title>Redirectare la situl FII</title>
</head>

<body>
  <applet code="FIISite.class" width="10" height="10">
  </applet>
</body>
</html>
```

iar appletul va avea următorul cod:

```
import java.applet.Applet;
import java.applet.AppletContext;
import java.net.MalformedURLException;
import java.net.URL;

public class FIISite extends Applet {
  public void start() {
    AppletContext context = getAppletContext();
    try {
      context.showDocument(new URL("http://www.infoiasi.ro/"));
    } catch (MalformedURLException e) {
      System.err.println("Adresa eronata"); }
  }
}
```

De obicei, această facilitate este folosită în urma interacţiunii utilizatorului cu suprafaţa de afişare a appletului.

7.2.4. Comunicarea dintre appleturi

Două appleturi pot comunica între ele dacă sunt pe acelaşi server şi sunt integrate în aceeaşi pagină Web. Modalitatea de comunicare este simplă: unul dintre appleturi obţine contextul său (care este şi al celuilalt applet, întrucât fac parte din aceeaşi pagină Web) şi poate obţine o referinţă la celălalt applet prin apelul metodei `getApplet()` din context, după care se pot apela metodele publice ale acestuia din urmă. Prin apelul de către un applet al metodelelor publice ale altui applet se poate realiza comunicarea.

Exemplul 7.2.7. Fişierul `comunicare.html` conţine două appleturi: `receptor` şi `emitator`. Appletul `emitator` va transmite un mesaj appletului `receptor`, iar acesta îl va afişa. Fişierul XHTML este următorul:

```
<html>
<head>
  <title>Comunicare intre appleturi</title>
</head>

<body>
  <applet name="receptor" code="AppletReceptor.class"
    width="400" height="60">
  </applet>
  <hr />
  <applet name="emitator" code="AppletEmitator.class"
    width="400" height="60">
  </applet>
```

```
</body>
</html>
```

Appletul `receptor` are următorul cod:

```
import java.applet.Applet;
import java.awt.Graphics;

public class AppletReceptor extends Applet {
  private String mesaj = "Nu am primit nici un mesaj!";

  public void primeste(String mesaj) {
    this.mesaj = mesaj;
    repaint();
  }
  public void paint(Graphics g) {
    g.drawString("Receptor",10,20);
    g.drawString(mesaj,10,50);
  }
}
```

Atunci când primeşte un mesaj, se va modifica data membră privată `mesaj`, după care se va invoca metoda `repaint()`, care la rândul ei va apela indirect metoda `paint()`. Acest lucru este util pentru afişarea mesajului primit.

Appletul `emitator` conţine următorul cod Java:

```
import java.applet.Applet;
import java.applet.AppletContext;
import java.awt.Graphics;

public class AppletEmitator extends Applet {
  private String msg = "";

  public void start() {
    // obtinerea contextului
    AppletContext context = getAppletContext();
    int i=0;
    AppletReceptor receptor = null;
    do {
      receptor = (AppletReceptor) context.getApplet("receptor");
      i++;
      System.out.println(i);
      try {
        Thread.sleep(100);
      } catch (InterruptedException e) {}
    } while(i<1000 && receptor == null);
    if (receptor == null)
```

```
        msg = "Nu am gasit receptorul!";
      else {
        receptor.primeste("Salut prietene!");
        msg = "Mesajul a fost trimis!";
      }
      repaint();
    }
    public void paint(Graphics g) {
      g.drawString("Emitator", 10, 20);
      g.drawString(msg, 10, 50);
    }
  }
```

Mai întâi se obţine contextul appletului după care se încearcă, eventual de mai multe ori, obţinerea unei referinţe la appletul receptor. Se încearcă de mai multe ori atunci când pagina Web nu este încărcată complet şi appletul receptor încă nu este vizibil. Când s-a terminat de încărcat, atunci vom putea obţine referinţa dorită şi vom apela metoda `primeste()` a appletului receptor. Acesta va primi mesajul şi-l va afişa.

În exemplul ilustrat anterior, comunicarea este unidirecţională, de la appletul emiţător la cel receptor. Comunicarea se poate realiza şi în ambele sensuri, dacă emiţătorul va conţine o metodă publică de primire a mesajului, iar receptorul va emite, la rândul său, mesaje. Se pot schimba mai multe mesaje (sincrone sau asincrone) între appleturi.

7.2.5. Redarea sunetelor şi imaginilor

Programele Java pot avea ca ieşire elemente grafice şi/sau sunete. Informaţiile de grafică şi sunet sunt memorate în fişiere. Cele mai cunoscute formate de fişiere grafice sunt GIF (*Graphics Interchange Format*), JPEG (*Joint Photographic Experts Group*) şi PNG (*Portable Network Graphics*). Fişierele GIF au extensia `.gif` şi folosesc pentru reprezentarea unui pixel cel mult 8 biţi. Fişierele JPEG au extensia `.jpg` şi utilizează pentru reprezentarea unui pixel 24 de biţi. JPEG este mai bun şi ca rezoluţie decât GIF şi are o compresie superioară a informaţiei grafice. Fişierele PNG au extensia `.png`, au 8 biţi pentru fiecare pixel şi oferă o mai bună compresie decât GIF-urile.

Este recomandată utilizarea fişierelor GIF sau PNG atunci când imaginile conţin mai puţin de 256 de culori.

7.2.5.1. *Afişarea conţinutului grafic*

Presupunem că dorim să vizualizăm o imagine grafică conţinută în fişierul `pictura.gif`. Pentru acest lucru trebuie să parcurgem două etape:

1. obţinerea unei referinţe la imagine;
2. încărcarea şi afişarea imaginii.

În prima etapă se invocă una dintre cele două forme ale metodei `getImage()` din clasa `Applet`:

- `public Image getImage(URL urlImagine)`
- `public Image getImage(URL urlDirector, String numeFisier)`

unde `urlImagine` este URL-ul absolut la imagine, `urlDirector` este URL-ul unui director care conţine imaginea, iar `numeFisier` este numele fişierului grafic, eventual un URL relativ faţă de adresa directorului specificat.

Exemple:

- `Image image = getImage(getDocumentBase(),"pictura.gif");`
- `Image image = getImage(new URL("http://picturi.polirom.ro/pictura.gif"));`

În primul caz, imaginea trebuie să fie plasată în acelaşi director cu documentul XHTML care conţine appletul. În al doilea caz, imaginea trebuie să aibă URL-ul specificat.

Încărcarea şi afişarea imaginii se realizează cu metoda `drawImage()` din clasa `Graphics`. Cel mai frecvent utilizate forme ale acestei metode sunt:

- `abstract boolean drawImage(Image imagine, int x, int y, Color bgColor, ImageObserver observator)`
- `abstract boolean drawImage(Image imagine, int x, int y, ImageObserver observator)`
- `abstract boolean drawImage(Image imagine, int x, int y, int latime, int inaltime, Color bgColor, ImageObserver observator)`
- `abstract boolean drawImage(Image imagine, int x, int y, int latime, int inaltime, ImageObserver observator)`

Parametrul `imagine` este o referinţă la imaginea care se va afişa (cea obţinută la pasul 1); `x` şi `y` reprezintă coordonatele unde se va afişa imaginea (respectiv colţul din stânga-sus); `latime` şi `inaltime` stabilesc lăţimea şi respectiv înălţimea imaginii (implicite sunt dimensiunile reale ale imaginii); `bgColor` reprezintă culoarea de fundal utilizată şi are efect doar dacă imaginea are porţiuni transparente; `observator` este un obiect care semnalează cât din imagine s-a încărcat şi, de obicei, este obiectul curent, adică `this`. Metoda se termină imediat chiar dacă imaginea nu s-a încărcat complet. Se returnează `true`, dacă imaginea s-a încărcat complet şi `false`, în caz contrar.

Atât `Image`, cât şi `Graphics` sunt clase abstracte. `ImageObserver` este o interfaţă din pachetul `java.awt.image` care furnizează informaţii despre încărcarea imaginii. Clasa `Applet` implementează interfaţa `ImageObserver`.

Exemplu de utilizare a metodei `drawImage()`:

```
boolean b = g.drawImage(image, 20, 20, 100, 100, this);
```

Exemplul 7.2.8. Afişarea unei imagini.

```
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.Image;

public class InserareImagine extends Applet {
  private Image imagine;
  public void init() {
    // obtinerea referintei la imagine
    imagine = getImage(getDocumentBase(), "imagine.jpg");
  }

  public void paint(Graphics g) {
    // desenarea imaginii
    boolean b = g.drawImage(imagine, 20, 20, 100, 100, this);
  }
}
```

Dacă imaginea nu va fi găsită, atunci aceasta nu va fi inclusă. De remarcat este faptul că nu apare nici un mesaj de eroare în acest caz.

7.2.5.2. *Redarea sunetelor*

Ca şi imaginile grafice, sunetele sunt memorate în fişiere. Sunt suportate următoarele formate: AIFF, AU, WAV, MIDI, RMF. Tipul de fişier cel mai utilizat este fişierul audio, cu extensia `.au`. Ca şi la imagini, mai întâi trebuie să obţinem o referinţă la un fişier audio, iar acest lucru se realizează cu metoda `getAudioClip()` din clasa `Applet`:

- `public AudioClip getAudioClip(URL urlAudioClip)`
- `public AudioClip getAudioClip(URL urlDirector, String numeFisier)`

unde parametrii au aceeaşi semnificaţie ca la metoda `getImage()` prezentată în secţiunea anterioară, cu deosebirea că acum ne referim la un fişier audio, iar tipul returnat este `AudioClip`.

Interfaţa `AudioClip` se găseşte în pachetul `java.applet` şi conţine următoarele metode:

Prototipul metodei	Descrierea metodei
public void **loop**()	Porneşte difuzarea clipului audio de mai multe ori.
public void **play**()	Difuzează conţinutul audio o singură dată.
public void **stop**()	Opreşte difuzarea clipului.

Programul Java continuă să se execute în timp ce sunetul este difuzat.

Exemplul 7.2.9. Appletul pune la dispoziţie trei butoane, câte unul pentru fiecare metodă din interfaţa `AudioClip`:

```
import java.applet.Applet;
import java.applet.AudioClip;
import java.awt.Button;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class Audio extends Applet
  implements ActionListener {

  private Button play, loop, stop;
  private AudioClip sound;

  public void init() {
    // crearea butoanelor
    play = new Button("Play");
    add(play);
    play.addActionListener(this);
    loop = new Button("Loop");
    add(loop);
    loop.addActionListener(this);
    stop = new Button("Stop");
    add(stop);
    stop.addActionListener(this);
    // obtinerea referintei la clip
    sound = getAudioClip(getDocumentBase(), "clip.wav");
  }

  public void actionPerformed(ActionEvent event) {
    // in functie de butonul apasat
    // se va apela metoda corespunzatoare
    if (event.getSource() == play)
      sound.play();
    if (event.getSource() == loop)
      sound.loop();
    if (event.getSource() == stop)
      sound.stop();
  }
}
```

Dacă fişierul `clip.wav` nu a fost găsit, atunci apăsarea butoanelor nu va avea nici un efect.

7.2.6. Transformarea unui applet într-o aplicaţie de sine stătătoare

Se poate întâmpla, de multe ori, să dorim ca un applet să fie şi aplicaţie de sine stătătoare. Pentru aceasta va trebui să adăugăm metoda publică şi statică `main()` cu rolul de a crea o fereastră în care va fi inclus respectivul applet. Comentariile din al doilea program Java din exemplul următor indică etapele ce trebuie urmate ca appletul să fie şi aplicaţie de sine stătătoare.

Exemplul 7.2.10. Fie următorul applet simplu pentru care dorim să fie şi aplicaţie de sine stătătoare în acelaşi timp:

```
import java.applet.Applet;
import java.awt.*;

public class AppletSimplu extends Applet {
  public void paint(Graphics g) {
    g.drawString("Applet simplu.", 10, 20);
  }
}
```

Appletul nu face altceva decât să afişeze un mesaj. Adăugând metoda `main()`, appletul va deveni şi aplicaţie de sine stătătoare:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class AppletSimplu extends Applet {
  public void paint(Graphics g) {
    g.drawString("Applet simplu.", 10, 20);
  }

  // cream o instanta a appletului
  static final AppletSimplu applet = new AppletSimplu();

  public static void main(String args[]) {
    // cream o fereastra
    Frame fereastra = new Frame("Applet simplu");
    // adaugam appletul la fereastra
    fereastra.add(applet, BorderLayout.CENTER);
    // stabilim dimensiunea si pozitionarea ferestrei
    fereastra.setBounds(100, 100, 200, 70);
    // stabilim un ascultator (inchidere)
    // pentru operatiile cu fereastra
```

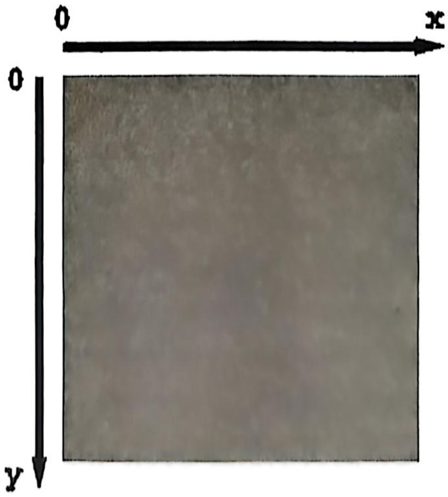
```
        fereastra.addWindowListener(new WindowAdapter() {
          public void windowClosing(WindowEvent e) {
            // distrugem appletul
            applet.destroy();
            // inchidem fereastra
            Window win = e.getWindow();
            win.dispose();
          }
        });
        // initializam appletul
        applet.init();
        // pornim appletul
        applet.start();
        // activam fereastra
        fereastra.setVisible(true);
    }
}
```

În aplicaţia Java de mai sus a trebuit să mai declarăm o clasă anonimă care să se ocupe de închiderea ferestrei. Informaţii amănunţite referitoare la lucrul cu ferestre se găsesc în secţiunea 7.6.

7.3. Grafica în Java

Biblioteca AWT (engl. *Abstract Window Toolkit*) conţine toate clasele pentru crearea interfeţelor cu utilizatorul şi pentru desenarea graficelor sau imaginilor.

Clasa abstractă `Graphics` face parte din pachetul `java.awt` şi se referă la ecranul grafic Java. Grafica Java se bazează pe pixeli şi pe un sistem ortogonal de axe. Astfel, avem axa Ox pe orizontală şi Oy pe verticală. Punctul de coordonate (0,0) se află în colţul din stânga-sus al suprafeţei grafice.



Axele de coordonate în Java

De obicei, dimensiunea unei suprafeţe grafice pentru un applet este de 300 de pixeli lungime (pe Ox) şi de 200 de pixeli înălţime (pe Oy).

Cele mai utilizate metode ale clasei `Graphics` sunt explicate în tabelul de mai jos:

Prototipul metodei	Descrierea metodei
`public abstract void` **`drawString`**`(String str, int x, int y)`	Desenează şirul de caractere `str`, începând cu poziţia `(x,y)`. Se utilizează fontul şi culoarea curentă.
`public abstract void` **`drawLine`**`(int x1, int y1, int x2, int y2)`	Desenează o linie între punctele de coordonate `(x1,y1)` şi `(x2,y2)` utilizând culoarea curentă.
`public abstract void` **`drawRect`**`(int x, int y, int lungime, int inaltime)`	Desenează chenarul unui dreptunghi având colţul din stânga-sus la coordonata `(x,y)`, iar cel din dreapta-jos la `(x+lungime,y+inaltime)`.
`public abstract void` **`drawOval`**`(int x, int y, int lungime, int inaltime)`	Desenează chenarul unei elipse în dreptunghiul dat de parametrii metodei (care au aceeaşi semnificaţie ca aceia ai metodei `drawRect()`).
`public abstract void` **`drawArc`**`(int x, int y, int lungime, int inaltime, int unghiInceput, int unghiSfarsit)`	Desenează un arc de elipsă încadrat într-un dreptunghi dat de primii patru parametri. `unghiInceput` este un unghi de început, iar `unghiSfarsit` este un unghi cu care se termină arcul şi este relativ la cel de început. Măsura unghiurilor este dată în grade hexazecimale.

Funcţiile de „umplere" sunt `fillRect()`, `fillArc()`, `fillOval()` şi au aceiaşi parametri ca funcţiile echivalente de desenare (de tip `draw...`), diferenţa constând în colorarea interiorului utilizând culoarea curentă.

Pentru a lucra cu altă culoare în afară de cea neagră, va trebui să utilizăm obiecte din clasa `java.awt.Color`. Aceasta pune la dispoziţie pentru culorile des folosite mai multe date membre statice de tip `Color`. Acestea sunt: `black` (negru), `blue` (albastru), `cyan`, `darkGray` (gri închis), `gray` (gri), `green` (verde), `lightGray` (gri deschis), `magenta`, `orange` (oranj), `pink` (roz), `red` (roşu), `white` (alb) şi `yellow` (galben). Pentru a obţine o altă culoare, vom utiliza constructorul:

```
public Color(int r, int g, int b);
```

care are ca parametri codul RGB (*Red-Green-Blue*). Aceştia au valori cuprinse între 0 şi 255 şi indică gradul în care se combină cele trei culori (roşu, verde şi respectiv albastru) pentru obţinerea noii culori. Dacă dorim să obţinem o culoare care este transparentă într-o anumită proporţie, atunci vom utiliza constructorul:

```
public Color(int r, int g, int b, int transparenta);
```

unde primii trei parametri formează codul RGB al culorii, iar ultimul are valoarea cuprinsă tot între 0 şi 255, 0 desemnând o transparenţă totală, iar 255 stabilind o culoare opacă.

Pentru stabilirea culorii de fundal a unui applet putem apela metoda `setBackground()` moştenită de `Applet` din clasa `Component`, iar culoarea pentru desenare, cu metoda `setColor()` din clasa `Graphics`. Ambele metode au un singur parametru de tip `Color`.

Exemplul 7.3.1. Appletul va utiliza metodele explicate anterior. Se vor evidenţia dreptunghiurile circumscrise arcelor.

```
import java.applet.Applet;
import static java.awt.Color.*;
import java.awt.Graphics;

public class DesenSimplu extends Applet {
  public void paint(Graphics g) {
    g.drawString("Exercitiu de grafica", 10, 15);
    g.drawRect(30, 30, 80, 40);
    g.drawOval(120, 30, 50, 50);
    g.setColor(red);
    g.fillRect(30, 100, 80, 40);
    g.fillOval(120, 100, 50, 50);
    g.drawLine(30, 160, 130, 170);
    g.drawArc(30, 180, 30, 50, 60, 90);
    g.fillArc(120, 180, 50, 50, 60, 40);
    g.setColor(black);
    g.drawRect(30, 180, 30, 50);
    g.drawRect(120, 180, 50, 50);
    setBackground(lightGray);
  }
}
```

Exemplul 7.3.2. Appletul va desena două triunghiuri isoscele şi va calcula aria acestora.

```
import java.applet.Applet;
import java.awt.Color;
import java.awt.Graphics;

public class TriunghiIsoscel extends Applet {
  public void paint(Graphics g) {
    g.setColor(Color.blue);
    deseneaza_triunghi(g, 80, 200, 100, 110);
    g.drawString("Aria triunghiului este " +
      aria_triunghi(100, 110), 135, 130);
    g.setColor(Color.magenta);
    deseneaza_triunghi(g, 125, 220, 60, 70);
    g.drawString("Aria triunghiului este " +
      aria_triunghi(60, 70), 150, 180);
  }
```

```
    private void deseneaza_triunghi(Graphics g, int bottomX,
        int bottomY, int baza, int inaltime) {
      int rightX = bottomX + baza;
      int topX = bottomX + baza/2;
      int topY = bottomY - inaltime;
      g.drawLine(bottomX, bottomY, rightX, bottomY);
      g.drawLine(rightX, bottomY, topX, topY);
      g.drawLine(topX, topY, bottomX, bottomY);
    }

    private int aria_triunghi(int baza, int inaltime) {
      int aria = (baza * inaltime)/2;
      return aria;
    }
}
```

Observaţie! Toate metodele private (definite de noi) sunt declarate şi apelate, în schimb, funcţia `paint()` este doar declarată, şi nu apelată! Navigatorul Web sau `appletviewer`-ul iniţializează appletul şi apelează funcţia `paint()`. De aceea, aceasta nu poate fi `private`, ci `public`.

Exemplul 7.3.3. Următorul applet animează o minge pe ecran. Mingea se află într-un dreptunghi şi ricoşează din pereţii acestuia.

```
import java.applet.Applet;
import java.awt.Color;
import java.awt.Graphics;

public class Minge extends Applet implements Runnable {
  private int x = 7, xChange = 7;
  private int y = 2, yChange = 2;
  private int diameter = 10;
  private int rectLeftX = 0, rectRightX = 100;
  private int rectTopY = 0, rectBottomY = 100;
  private boolean ok;

  public void init() {
    ok = true;
  }

  public void paint(Graphics g) {
   g.drawRect(rectLeftX, rectTopY, rectRightX - rectLeftX + 10,
       rectBottomY - rectTopY + 10);
    // desenam cu culoarea fundalului
    // pentru a acoperi mingea anterior desenata
```

```
      g.setColor(getBackground());
      // se sterge mingea
      g.fillOval(x, y, diameter, diameter);
      // daca se intalneste peretele se schimba directia de miscare
      if (x + xChange <= rectLeftX)
        xChange = - xChange;
      if (x + xChange >= rectRightX)
        xChange = - xChange;
      if (y + yChange <= rectTopY)
        yChange = - yChange;
      if (y + yChange >= rectBottomY)
        yChange = - yChange;
      // se modifica noua pozitie a mingii
      x = x + xChange;
      y = y + yChange;
      // se stabileste culoarea rosie pentru desenare
      g.setColor(Color.red);
      // se deseneaza mingea
      g.fillOval(x, y, diameter, diameter);
    }

    // se apeleaza la distrugerea appletului
    // si are ca efect terminarea metodei paint()
    public void destroy() {
      ok = false;
    }

    public void start() {
      ok = true;
      new Thread(this).start();
    }

    public void stop() {
      ok = false;
    }

    public void run() {
      while (ok) {
        repaint();
        try {
          Thread.sleep(40);
        } catch (InterruptedException e) {}
      }
    }
  }
```

7.4. Componente şi evenimente

O *componentă* (grafică) este o entitate grafică care poate fi adăugată la suprafaţa de afişare a unui program sau applet Java. Utilizatorul poate interacţiona cu programul prin intermediul acestor componente. La acţionarea asupra unei astfel de componente se emite un eveniment. Pentru ca la apariţia unui eveniment să-i dăm programului o anumită funcţionalitate, va trebuie să ataşăm un „ascultător" care să implementeze efectul dorit.

Un *eveniment* este o acţiune impusă de către utilizator. De exemplu, acţiuni sunt apăsarea unei taste sau a unui buton de mouse de către utilizator, manevrarea unei bare de derulare (*Scroll*), marcarea unei opţiuni dintr-un meniu.

Aşadar, în loc să ne imaginăm un program ca un şir de instrucţiuni, putem să ne închipuim secţiunile programului ca fiind activate de un eveniment (se mai spune că programul este „condus de evenimente").

Un program „condus de evenimente" (engl. *event-driven*) trebuie să rezolve problema evenimentelor (engl. *event-handling*). Detectarea evenimentelor şi rezolvarea sarcinii cerute se numeşte *rezolvarea evenimentului*. Programele care utilizează acest stil de interfaţă cu utilizatorul se numesc „conduse de evenimente".

7.4.1. Bucla evenimentelor

Nu trebuie să codificăm bucla evenimentelor şi nu trebuie să facem recunoaşterea evenimentelor la nivelul de jos. În Java, creăm o secţiune de programe (metode) pe care sistemul Java le apelează automat când un eveniment are loc, în plus dându-le detalii despre eveniment. Apoi, aceste detalii (de exemplu, apăsarea butonului-stânga al mouse-ului) sunt utile pentru a asigura că procesul cerut are loc.

În Java, evenimentele sunt clasificate. De exemplu, barele de derulare (engl. *scrollbars*) sunt, de obicei, utilizate pentru modificarea valorii sau vizualizarea unei alte părţi a ecranului. Butoanele, în schimb, prin apăsarea lor, sunt folosite pentru iniţierea sau terminarea unei activităţi pentru îndeplinirea unei activităţi. Obiectele (interfeţele) care se ocupă de aceste lucruri se numesc ascultători (engl. *listeners*) şi au sufixul `Listener`. Pentru butoane există clasa `ActionListener`, iar pentru barele de defilare, `AdjustmentListener`.

Fiecare ascultător conţine câte o metodă pentru fiecare tip de acţiune care ne interesează (o metodă pentru derularea barei de defilare, o metodă pentru un clic etc.). În plus, trebuie să desemnăm un ascultător pentru componenta (engl. *scrollbar*) care produce evenimente, iar metodele acestuia vor fi apelate automat când apare evenimentul corespunzător. Prin redefinirea metodelor ascultătorului putem conferi componentei noastre funcţionalitatea dorită.

Fiecărui eveniment îi corespunde o clasă care are sufixul `Event`. Ascultătorii şi evenimentele sunt clase care fac parte din pachetul `java.awt.event`. Tot în acest pachet se găsesc unele clase care implementează interfeţele ascultătorilor şi nu au nici un efect la primirea evenimentelor. Aceste clase se mai numesc adaptori (engl. *adapters*).

Exemplul 7.4.1. Programul Java de mai jos (condus de evenimente) afişează numărul corespunzător deplasării unei bare de defilare.

```
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.Scrollbar;
import java.awt.event.AdjustmentEvent;
import java.awt.event.AdjustmentListener;

public class Slider extends Applet
    implements AdjustmentListener {

   private Scrollbar slider;
   private int sliderValue = 0;

   public void init() {
      // se creeaza o bara de derulare
      slider = new Scrollbar(Scrollbar.HORIZONTAL,
         0, 1, 0, 100);
      // se adauga bara la suprafata de desenare
      // a appletului
      add(slider);
      // se stabileste un ascultator pentru bara
      slider.addAdjustmentListener(this);
   }

   public void paint(Graphics g) {
      g.drawString("Valoarea curenta este " +
         sliderValue, 100, 70);
   }

   public void adjustmentValueChanged(AdjustmentEvent e) {
      // se salveaza pozitia barei de defilare
      sliderValue = slider.getValue();
      // se invoca redesenarea appletului
      repaint();
   }
}
```

Clasa `Slider` implementează interfaţa `AdjustmentListener`, care este un ascultător pentru evenimentele specifice barelor de derulare. În metoda de iniţializare se creează o bară de derulare care este adăugată la suprafaţa de desenare a appletului. La finalul metodei se stabileşte un ascultător pentru bara de defilare prin apelul metodei `addAdjustmentListener()`. Aceasta are ca parametru un obiect care implementează interfaţa `AdjustmentListener`. Interfaţa posedă metoda `adjustmentValueChanged()`, care este apelată ori de câte ori bara de defilare este utilizată.

Acest program permite doar manipularea barei de defilare. De fiecare dată când mişcăm bara de defilare, metoda `adjustmentValueChanged()` este invocată. În cazul nostru, metoda `adjustmentValueChanged()` copiază noua valoare din bara de defilare în variabila `sliderValue`, apoi redesenează suprafaţa appletului.

Metoda `void adjustmentValueChanged(AdjustmentEvent e)` este utilizată pentru procesarea evenimentelor barei de defilare. Când un astfel de eveniment apare, această metodă este automat apelată.

Pentru crearea unei bare de derulare s-a instanţiat clasa `java.awt.Scrollbar` apelându-se la constructorul:

```
public Scrollbar(int orientare, int valoare, int vizibilitate,
   int minim, int maxim)
```

Parametrul `orientare` indică orientarea barei de defilare: orizontală (caz în care se utilizează data membră statică `Scrollbar.HORIZONTAL`) sau verticală (`Scrollbar.VERTICAL`). Al doilea parametru indică valoarea iniţială pe care o va avea bara, iar al treilea se referă la dimensiunea dreptunghiului din bara de derulare a cărui glisare duce la modificarea valorii barei. Parametrii `minim` şi `maxim` se referă la marginile domeniului în care ia valori deplasamentul barei de defilare.

Exemplul 7.4.2. Programul afişează o bară de defilare verticală, iar odată cu modificarea sa se schimbă şi nivelul de umplere a unui dreptunghi.

```
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.Scrollbar;
import java.awt.event.AdjustmentEvent;
import java.awt.event.AdjustmentListener;

public class VSlider extends Applet
    implements AdjustmentListener {
  private Scrollbar slider;
  private int sliderValue;

  public void init() {
    // crearea unei bare de derulare verticale
    slider = new Scrollbar(Scrollbar.VERTICAL,
       0, 1, 0, 101);
    // adaugarea barei la applet
    add(slider);
    // stabilirea unui ascultator pentru bara
    slider.addAdjustmentListener(this);
  }

  public void paint(Graphics g) {
    // se afiseaza in bara de stare a navigatorului
```

```
        showStatus("Valoarea barei Scroll este " +
            sliderValue);
        // desenarea dreptunghiului
        g.drawRect(40, 80, 60, 100);
        // umplerea acestuia in functie de deplasarea
        // barei de derulare
        g.fillRect(40, 80, 60, sliderValue);
    }

    public void adjustmentValueChanged(AdjustmentEvent e) {
        sliderValue = slider.getValue();
        repaint();
    }
}
```

Acest exemplu este similar cu exemplul 7.4.1. Observăm în plus că, la desenare, metoda `fillRect()` umple un dreptunghi care are înălţimea egală cu valoarea lui `sliderValue`, adică cu valoarea derulării barei verticale.

Programul este condus de evenimente, deoarece aşteaptă utilizarea barei de defilare. Evenimentele sunt procesate de metoda `adjustmentValueChanged()`. Aceasta are un singur parametru de tip `AdjustmentEvent` care conţine detalii despre eveniment. În programul nostru există doar un eveniment (mişcarea barei de defilare).

O bară de derulare poate avea doar valori întregi în domeniul stabilit. Se poate întâmpla să avem nevoie de valori care să fie numere zecimale, de exemplu, 0.1, 0.15, 0.2... Pentru aceasta vom scala bara de defilare.

Vom considera un exemplu de convertire a inchilor (din domeniul {0, ..., 10}) la centimetri, unde 1 inch = 2,54 cm. Putem seta bara de defilare la domeniul 0..10, dar pentru mai multă precizie vom crea bara de defilare în domeniul {0, ..., 100} şi vom scala aceşti întregi către reali rezultând paşi de 0.1 inchi.

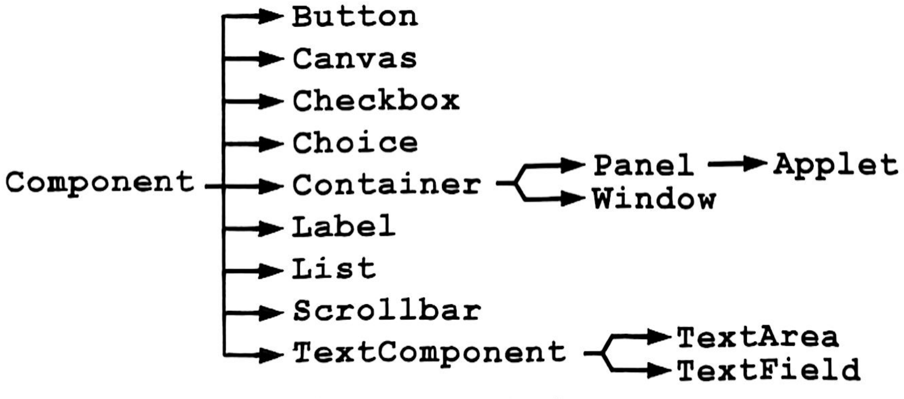
Exemplul 7.4.3. Conversia din inchi în centimetri.

```
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.Scrollbar;
import java.awt.event.AdjustmentEvent;
import java.awt.event.AdjustmentListener;

public class Inch2Centimetri extends Applet
    implements AdjustmentListener {

  private Scrollbar slider;
  private float sliderValue;

  public void init() {
    slider = new Scrollbar(Scrollbar.HORIZONTAL,
        0, 1, 0, 101);
    add(slider);
    slider.addAdjustmentListener(this);
  }

  public void paint(Graphics g) {
    // transformarea din inchi in centimetri
    float cm = sliderValue * 2.54f;
    g.drawString("Inchi = " + sliderValue +
        " Cm=" + cm, 100, 100);
  }

  public void adjustmentValueChanged(AdjustmentEvent e) {
    // scalarea din numere intregi in numere zecimale
    sliderValue = (float) slider.getValue()/10;
    repaint();
  }
}
```

7.4.2. Componente

O componentă este un obiect care are reprezentare grafică, poate fi afişat pe ecran şi poate interacţiona cu utilizatorul.

Toate componentele, mai puţin meniurile, sunt derivate direct sau indirect din clasa `Component`. Metodele acestei clase se vor regăsi şi în clasele derivate şi vor stabili proprietăţile generale (respectiv obţinerea acestora) ale unei componente. Există componente care includ alte componente.


Component
Button
Canvas
Checkbox
Choice
Container
Panel
Applet
Window
Label
List
Scrollbar
TextComponent
TextArea
TextField

Ierarhia componentelor în AWT

Metodele cele mai semnificative din clasa `Component` sunt următoarele:

Prototipul metodei	Descrierea metodei
public Color **getBackground**()	Obţine culoarea de fundal.
public void **setBackground**(Color c)	Stabileşte culoarea fundalului.
public Color **getForeground**()	Întoarce culoarea de scriere sau de desenare.
public void **setForeground**(Color c)	Setează culoarea de scriere sau de desenare.
public Component void **getComponetAt**(int x, int y)	Întoarce componenta de la coordonatele (x,y, dacă aceasta este componenta curentă sau subcomponenta acestuia şi null, în caz contrar.
public int **getHeight**()	Obţine înălţimea componentei.
public int **getWidth**()	Obţine lăţimea componentei.
public void **setSize**(int latime, int inaltime)	Stabileşte lăţimea şi înălţimea componentei.
public void **setBounds**(int x, int y, int latime, int inaltime)	Stabileşte poziţia şi dimensiunile componentei (colţul din stânga-sus va fi la coordonatele (x,y)).
public int **getX**() public int **getY**()	Cele două metode determină coordonatele la care se află componenta.
public void **getName**()	Returnează numele componentei.
public void **setName**(String name)	Stabileşte numele componentei.
public Component **getParent**()	Întoarce componenta-părinte.
public boolean **hasFocus**()	Testează dacă obiectul curent are focusul.
public boolean **isShowing**()	Testează dacă componenta este activă pe ecran.
public boolean **isEnable**()	Testează dacă componenta este disponibilă.
public void **setEnable**(boolean b)	Stabileşte dacă componenta este disponibilă.
public void **paint**(Graphics g)	Desenează componenta. g este contextul grafic pentru desenare.
public void **paintAll**(Graphics g)	Desenează componenta şi subcomponentele sale.
public void **repaint**()	Redesenează componenta.

O importanţă aparte o au componentele care conţin alte componente. Acestea sunt denumite containere.

7.4.3. Containere

Un container este o componentă care poate conţine alte componente. Unui container îi corespunde clasa `java.awt.Container` care este derivată din clasa `Component`.

Clasa `Container` are doar un constructor implicit, iar metodele cele mai importante sunt explicate în tabelul de mai jos:

Prototipul metodei	Descrierea metodei
`public Component` **`add`**`(Component c)`	Adaugă componenta specificată la sfârşitul containerului. Returnează componenta adăugată.
`public Component` **`add`**`(Component c, int index)`	Adaugă componenta specificată în container, pe poziţia indicată. Returnează componenta adăugată.
`public void` **`add`**`(Component c, Object constrangeri)`	Adaugă componenta specificată la sfârşitul containerului şi stabileşte constrângeri pentru poziţionare. A se vedea subcapitolul 7.5.
`public void` **`add`**`(Object constrangeri, int index)`	Adaugă componenta specificată în container pe poziţia indicată şi stabileşte constrângeri pentru poziţionare. A se vedea subcapitolul 7.5.
`public Component` **`findComponentAt`**`(int x, int y)`	Returnează subcomponenta care se află la coordonata indicată sau `null` în caz de eşec.
`public Component` **`getComponentCount`**`()`	Returnează numărul de componente din container.
`public Component` **`getComponent`**`(int n)`	Întoarce componenta de pe poziţia n din container.
`public Component[]` **`getComponents`**`()`	Returnează un tablou cu toate componentele din container.
`public void` **`remove`**`(Component c)`	Elimină componenta specificată.
`public void` **`remove`**`(int index)`	Elimină componenta cu indexul indicat.
`public void` **`removeAll`**`()`	Elimină toate componentele din container.

Clasele cele mai importante care sunt derivate din clasa `Container` sunt `Panel` şi `Window` care vor fi prezentate ulterior în acest capitol.

7.4.4. Etichete

În continuare, vom vedea cum putem scrie un text lângă o bară de defilare (bineînţeles, fără a preciza coordonatele). Vom utiliza funcţia `add()` pentru a plasa o nouă componentă pe ecran. Implicit, se poziţionează componentele centrat, de la stânga la dreapta.

Vom introduce componenta `Label` care conţine text şi care se adaugă la fel ca barele de defilare.

Exemplul 7.4.4. Inserarea unei etichete într-un applet.

```
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;

public class AppletCuEtichete extends Applet
   implements AdjustmentListener {

  private Scrollbar bar1, bar2;
  private int bar1Value = 0;
  private int bar2Value = 0;

  public void init() {
    Label titlu1, titlu2;
    // crearea unei etichete
    titlu1 = new Label("Sus:");
    // adaugarea etichetei la applet
    add(titlu1);
    bar1 = new Scrollbar(Scrollbar.HORIZONTAL,
       0, 1, 0, 101);
    add(bar1);
    bar1.addAdjustmentListener(this);
    // crearea altei etichete
    titlu2 = new Label("       Jos:");
    // adaugarea etichetei la applet
    add(titlu2);
    bar2 = new Scrollbar(Scrollbar.HORIZONTAL,
       0, 1, 0, 101);
    add(bar2);
    bar2.addAdjustmentListener(this);
  }

  public void paint(Graphics g) {
    g.drawString("Valoarea de sus este " +
      bar1Value, 100, 100);
    g.drawString("Valoarea de jos este " +
      bar2Value, 100, 150);
  }

  public void adjustmentValueChanged(AdjustmentEvent e) {
    bar1Value = bar1.getValue();
```

```
        bar2Value = bar2.getValue();
        repaint();
    }
}
```

O etichetă nu este activă (în sensul că un clic asupra sa nu produce evenimente) şi de cele mai multe ori nu trebuie să ne referim la o etichetă mai târziu. Excepţie face cazul în care conţinutul etichetei se modifică în timpul execuţiei aplicaţiei. De aceea, domeniul unei etichete poate fi local în metoda de iniţializare.

Mai observăm că barele de defilare au acelaşi ascultător, iar dacă s-a modificat starea uneia dintre ele, va duce la apelarea aceleiaşi metode. Eventual, se poate defini pentru fiecare bară câte un ascultător.

7.4.5. Butoane

Unui buton îi corespunde un obiect din clasa `java.awt.Button`. Aceasta are doi constructori: unul fără parametri şi altul cu un parametru de tip `String`, care stabileşte textul ce va apărea pe buton. Aflarea şi setarea etichetei se realizează cu metodele publice `String getLabel()`, respectiv `void setLabel(String eticheta)`.

Ascultătorul unui buton trebuie să implementeze interfaţa `ActionListener`. Aceasta are o singură metodă care este apelată atunci când s-a apăsat butonul:

- `public void actionPerformed(ActionEvent e)`

Adăugarea unui ascultător la un buton se realizează cu metoda `addActionListener()` din clasa `Button`.

În clasa `java.lang.Math` există metoda statică `random()` ce returnează un număr aleatoriu de tip `double` cuprins în intervalul [0, 1).

Exemplul 7.4.5. Programul permite aruncarea a două zaruri şi testează dacă numerele obţinute sunt egale. Pentru o nouă aruncare a zarurilor se va apăsa pe un buton.

```
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;

public class Zaruri extends Applet implements
   ActionListener {

  private Button aruncaZar;
  private boolean arunca = false;

  public void init() {
    // crearea unui buton
    aruncaZar = new Button("Arunca");
    // adaugarea butonului la applet
    add(aruncaZar);
```

```
      // stabilirea ascultatorului
      aruncaZar.addActionListener(this);
   }

   // metoda este apelata la apasarea butonului
   public void actionPerformed(ActionEvent eveniment) {
      arunca = true;
      repaint();
   }

   public void paint(Graphics g) {
      int zar1, zar2;
      if (arunca) {
         zar1 = (int)(Math.random() * 6) + 1;
         zar2 = (int)(Math.random() * 6) + 1;
         g.drawString("Zarurile sunt " + zar1 + " si " +
                zar2, 20, 40);
         if (zar1 == zar2)
            g.drawString("Am castigat! Zarurile sunt egale",
                20, 60);
         else
            g.drawString("Am pierdut! Zarurile nu sunt egale",
                20, 60);
      }
   }
}
```

O altă modalitate de a obţine numere aleatorii este de a utiliza clasa `java.util.Random`. Aceasta posedă următoarele metode importante:

Prototipul metodei	Descrierea metodei
`public float` **`nextFloat`**`()`	Returnează următorul număr aleatoriu, distribuit uniform în intervalul [0, 1) (de tip `float`).
`public double` **`nextDouble`**`()`	Returnează următorul număr aleatoriu, distribuit uniform în intervalul [0, 1) (de tip `double`).
`public int` **`nextInt`**`()`	Returnează următorul număr de tip `int`. Toate numerele posibile de tip `int` sunt echiprobabile.
`public long` **`nextLong`**`()`	Returnează următorul număr de tip `long`. Toate numerele posibile de tip `long` sunt echiprobabile.
`public void` **`setSeed`**`(long seed)`	Setează numărul care stă la baza generatorului de numere aleatorii.

Exemplul 7.4.6. O altă modalitate de a genera numere aleatorii:

```
Random r = new Random();
```

```
float f = r.nextFloat();
g.drawString("Numarul aleatoriu este: " + f, 50, 50);
```

7.4.6. Câmpuri şi arii text

Citirea textelor în AWT se realizează prin intermediul câmpurilor text şi a ariilor text. Un câmp conţine o singură linie, în timp ce o arie text poate conţine mai multe linii.

7.4.6.1. *Clasa* `java.awt.TextComponent`

Câmpurile text sunt definite de clasa `java.awt.TextField`, iar ariile text de `java.awt.TextArea`. Ambele sunt derivate din clasa `java.awt.TextComponent`, derivată la rândul ei din clasa `Component`, deci va avea şi metodele acesteia. În tabelul de mai jos sunt prezentate câteva metode din clasa `TextComponent`:

Prototipul metodei	Descrierea metodei
`public void` **`addTextListener`**`(TextListener ascultator)`	Ataşează un ascultător de tip `TextListener`. După acest tabel este prezentată interfaţa `TextListener`.
`public Color` **`getBackground`**`()`	Obţine culoarea de fundal a componentei text.
`public void` **`setBackground`**`(Color c)`	Stabileşte culoarea de fundal a componentei.
`public int` **`getCaretPosition`**`()`	Returnează poziţia cursorului în cadrul textului.
`public void` **`setCaretPosition`**`(int pozitie)`	Mută cursorul la poziţia specificată.
`public boolean` **`isEditable`**`()`	Testează dacă textul din componentă poate fi modificat (returnează `true` în caz afirmativ).
`public void` **`setEditable`**`(boolean b)`	Stabileşte dacă textul poate fi modificat (caz în care `b` are valoare `true`) sau nu.
`public void` **`select`**`(int inceput, int sfarsit)`	Selectează textul începând cu poziţia `inceput` şi până la poziţia `sfarsit`.
`public void` **`selectAll`**`()`	Selectează tot textul din componentă.
`public int` **`getSelectionStart`**`()`	Întoarce poziţia de început a textului selectat.
`public int` **`getSelectionEnd`**`()`	Întoarce poziţia de sfârşit a textului selectat.
`public void` **`setSelectionStart`**`(int poz)`	Stabileşte poziţia de început a selecţiei.
`public void` **`setSelectionEnd`**`(int poz)`	Stabileşte poziţia de sfârşit a selecţiei.
`public String` **`getSelectionText`**`()`	Returnează textul selectat.
`public String` **`getText`**`()`	Returnează întregul text din componentă.
`public void` **`setText`**`()`	Stabileşte textul din componentă.

Unei componente text îi putem ataşa un ascultător de tip `TextListener`. Această interfaţă posedă o singură metodă care este apelată atunci când se modifică textul din componentă:

- `public void textValueChanged(TextEvent e)`

7.4.6.2. *Clasa* `java.awt.TextField`

Clasa `TextField` moşteneşte clasa `TextComponent` şi are următorii constructori (pe lângă cel fără parametri):

- `public TextField(int nrColoane)`
- `public TextField(String textInitial)`
- `public TextField(String textInitial, int nrColoane)`

unde parametrul `nrColoane` se referă la lăţimea componentei (exprimată în numărul de caractere), iar `textInitial` va fi textul care va apărea în cadrul câmpului.

Pentru a capta acţiunile utilizatorului cu privire la introducerea de date într-un câmp text, trebuie să asociem câmpului un ascultător de tip `ActionListener` sau `TextListener`.

În `java.lang.Integer` există funcţia:

- `public static int parseInt(String s)`

care converteşte şirul `s` de cifre zecimale într-un întreg. Se generează o excepţie de tipul `NumberFormatException`, dacă şirul nu conţine caractere valide.

Exemplul 7.4.7. Programul Java conţine un câmp text pentru citirea vârstei şi afişează un mesaj de permisiune de vot.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Varsta extends Applet
   implements TextListener {
  private TextField campVarsta;
  private int varsta;

  public void init() {
    campVarsta = new TextField(10);
    add(campVarsta);
    campVarsta.addTextListener(this);
  }

  public void textValueChanged(TextEvent event) {
    try {
      varsta = Integer.parseInt(
               CampVarsta.getText());
    } catch (NumberFormatException e) {
            System.err.println("Exceptie de citire float");
        }
        repaint();
    }

    public void paint(Graphics g) {
        g.drawString("La varsta de " + varsta + " ani", 50, 50);
        if (varsta >= 18)
            g.drawString("puteti vota", 50, 70);
        else
            g.drawString("nu puteti vota", 50, 70);
    }
}
```

7.4.6.3. *Clasa* **`java.awt.TextArea`**

Clasa `TextArea` moştenește clasa `TextComponent` (care a fost descrisă în secţiunea 7.4.6.1). Putem utiliza următorii constructori (în afara constructorului fără parametri):

- `public TextArea(int nrRanduri, int nrColoane)`
- `public TextArea(String textInitial)`
- `public TextArea(String textInitial, int nrRanduri, int nrColoane)`
- `public TextArea(String textInitial, int nrRanduri, int nrColoane, int bareDefilare)`

unde parametrii `nrRanduri` şi `nrColoane` se referă la numărul de rânduri, respectiv de coloane vizibile, `textInitial` va desemna textul care va fi în aria text (implicit este şirul vid), iar `bareDefilare` va specifica barele de derulare care vor fi disponibile. Pentru `bareDefilare`, clasa `TextArea` pune la dispoziţie următoarele date membre:

- `public static int SCROLLBARS_BOTH`
  ambele bare de defilare vor fi disponibile;
- `public static int SCROLLBARS_HORIZONTAL_ONLY`
  doar bara de defilare orizontală va fi disponibilă;
- `public static int SCROLLBARS_VERTICAL_ONLY`
  doar bara de defilare verticală va fi disponibilă;
- `public static int SCROLLBARS_NONE`
  nici o bară de defilare nu va fi disponibilă.

Toţi constructorii pot arunca excepţia `HeadlessException` în cazul în care nu sunt disponibile utilizatorului elementele de interacţiune necesare, cum ar fi tastatura, mouse-ul sau ecranul.

Cele mai utilizate metode din clasa `TextArea` sunt următoarele:

Prototipul metodei	Descrierea metodei
`public void` **`append`**`(String text)`	Adaugă textul indicat la sfârşitul textului din componentă.
`public void` **`insert`**`(String text, int pozitie)`	Inserează textul specificat la poziţia indicată.
`public void` **`replaceRange`**`(String text, int inceput, int sfarsit)`	Înlocuieşte textul cuprins între poziţiile `inceput` şi `sfarsit` cu textul specificat.
`public int` **`getColumns`**`()`	Întoarce numărul de coloane ale componentei.
`public int` **`getRows`**`()`	Întoarce numărul de rânduri ale componentei.
`public void` **`setColumns`**`(int nrColoane)`	Restabileşte numărul de coloane ale componentei.
`public void` **`setRows`**`(int nrRanduri)`	Restabileşte numărul de rânduri ale componentei.

Exemplul 7.4.8. Appletul va conţine o arie de text şi două butoane. Primul va duce la afişarea textului selectat, a poziţiei de început, respectiv de sfârşit a selecţiei, iar celălalt buton va afişa întregul text. Mesajele vor apărea în bara de stare a navigatorului.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class Editor extends Applet
    implements ActionListener {
  private TextArea text;
  private Button marcaj, afisareText;

  public void init() {
    // crearea unei arii de text cu
    // 10 randuri si 20 de coloane
    text = new TextArea(10, 20);
    // adaugarea ariei text la applet
    add(text);

    // crearea unui buton
    marcaj = new Button("Afisare Text Selectat");
    add(marcaj);
    marcaj.addActionListener(this);

    afisareText = new Button("Afisare Text");
    add(afisareText);
    afisareText.addActionListener(this);
    }

    // metoda de tratare a evenimentelor aparute
    // in urma apasarii celor doua butoane
    public void actionPerformed(ActionEvent e) {
      // in functie de butonul apasat
      if (e.getSource() == marcaj) {
        // se obtine textul selectat
        String textSelectat = text.getSelectedText();
        // se obtine pozitia de start a selectiei
        int initial = text.getSelectionStart();
        // se obtine pozitia de sfarsit a selectiei
        int ultim = text.getSelectionEnd();
        // se formeaza mesajul care va fi afisat in bara de
        // stare
        String sir = "Textul selectat incepe la pozitia " +
            initial + " pana la pozitia " + ultim +
            " si este: \"" + textSelectat + "\"";
        showStatus(sir);
      }
      if (e.getSource() == afisareText) {
        // se obtine intregul text
        String textul = text.getText();
        String sir = "Textul scris este: \"" + textul + "\"";
        showStatus(sir);
      }
    }
  }
```

Exemplul 7.4.9. Următorul applet trasează graficul funcţiei *f(x)* = *a* * *sin(x)* + *b*. Parametrii *a* şi *b* iau valori în intervalul [5, 5) şi sunt cu o singură zecimală.

```
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;

public class Grafic extends Applet implements AdjustmentListener {
  // obiectul grafic se va ocupa de desenarea propriu-zisa
  // a graficului functiei
  private ReprezentareGrafica grafic;
  // valorile fiecarui parametru vor fi date de starea
  // unei bare de defilare
  private Scrollbar aScrollbar, bScrollbar;

  public void init() {
      // se va instantia graficul
      grafic = new ReprezentareGrafica();

      // se va crea o eticheta pentru parametrul a
      Label aLabel = new Label("a:");
      add(aLabel);
      // se va crea o bara de defilare pentru parametrul a
      aScrollbar = new Scrollbar(Scrollbar.HORIZONTAL, 50, 1, 0,
         100);
      add(aScrollbar);
      aScrollbar.addAdjustmentListener(this);

      // se va crea o eticheta pentru parametrul b
      Label bLabel = new Label("b:");
      add(bLabel);
      // se va crea o bara de defilare pentru parametrul a
      bScrollbar = new Scrollbar(Scrollbar.HORIZONTAL, 50, 1, 0,
         100);
      add(bScrollbar);
      bScrollbar.addAdjustmentListener(this);
   }

   // functia de tratare a evenimentelor
   // care apar la barele de defilare
   public void adjustmentValueChanged(AdjustmentEvent ev) {
      int aVal = aScrollbar.getValue();
      int bVal = bScrollbar.getValue();
      grafic.seteazaParametri(aVal, bVal);
      repaint();
   }

   public void paint(Graphics g) {
      grafic.deseneaza(g);
   }
}

// clasa ReprezentareGrafica este responsabila cu
// desenarea graficului functiei
class ReprezentareGrafica {
  private final int xPixelStart = 30, xPixelEnd = 380, xOrigine =
     205;
  private final int yPixelStart = 30, yPixelEnd = 380, yOrigine =
     205;
   private final double xStart = -10d, xEnd = 10d;
   private final double scala =
    (xPixelEnd - xPixelStart) / (xEnd - xStart);
  private double a, b;
  private double aMin = -5d, aMax = 5d;
  private double bMin = -5d, bMax = 5d;

  // functia noastra
  private double functie(double x, double a, double b) {
    return a*x*Math.sin(x)+b;
  }

  // scalarea pe orizontala
  private double scalaX(int xPixel) {
    double valoare = (xPixel - xOrigine) / scala;
    return valoare;
  }

  // scalarea pe verticala
  private int scalaY(double y) {
    int pixelCoord;
    pixelCoord = (int) Math.round(-y * scala) + yOrigine;
    return pixelCoord;
  }

  // modificarea parametrilor a si b
  public void seteazaParametri(int aVal, int bVal) {
    a = (aMax + aMin)/2 + (aVal - 50) * (aMax - aMin) / 100;
    b = (bMax + bMin)/2 + (bVal - 50) * (bMax - bMin) / 100;
  }

  // desenarea axei Ox
  private void abscisa(Graphics g) {
    g.drawLine(xPixelStart, yOrigine, xPixelEnd, yOrigine);
    g.drawLine(xPixelEnd - 5, yOrigine - 5, xPixelEnd, yOrigine);
    g.drawLine(xPixelEnd - 5, yOrigine + 5, xPixelEnd, yOrigine);
    int i, iScala;
    for (i = 1; i * (int) scala < xOrigine - xPixelStart; i++) {
      iScala = i * (int) scala;
      g.drawLine(xOrigine + iScala, yOrigine,
                 xOrigine + iScala, yOrigine - 2);
      g.drawLine(xOrigine - iScala, yOrigine,
                 xOrigine - iScala, yOrigine - 2);
    }
  }

  // desenarea axei Oy
```

```
    private void ordonata(Graphics g) {
      g.drawLine(xOrigine, yPixelEnd, xOrigine, yPixelStart);
      g.drawLine(xOrigine - 5, yPixelStart + 5, xOrigine,
         yPixelStart);
      g.drawLine(xOrigine + 5, yPixelStart + 5, xOrigine,
         yPixelStart);
      int i, iScala;
      for (i = 1; i * (int) scala < yOrigine - yPixelStart; i++) {
        iScala = i * (int) scala;
        g.drawLine(xOrigine, yOrigine - iScala,
                   xOrigine - 2, yOrigine - iScala);
        g.drawLine(xOrigine, yOrigine + iScala,
                   xOrigine - 2, yOrigine + iScala);
      }
    }

    // desenarea propriu-zisă a graficului
    public void deseneaza(Graphics g) {
      double x, y, nextX, nextY;
      int xPixel, yPixel, nextXPixel, nextYPixel;
      g.drawString("  a="+a+"  b="+b, 30, 60);
      abscisa(g);
      ordonata(g);
      for (xPixel = xPixelStart; xPixel < xPixelEnd; xPixel ++) {
        x = scalaX(xPixel);
        y = functie(x, a, b);
        yPixel = scalaY(y);
        nextXPixel = xPixel + 1;
        nextX = scalaX(nextXPixel);
        nextY = functie(nextX, a, b);
        nextYPixel = scalaY(nextY);
        g.drawLine(xPixel, yPixel, nextXPixel, nextYPixel);
      }
    }
}
```

Se observă din metoda `deseneaza()` că graficul funcţiei este construit din segmente de dreaptă.

7.4.7. Butoane de selecţie (*checkbox* şi radio)

Clasa care corespunde unui câmp de tip *checkbox* este `java.awt.Checkbox`. Aceasta este o componentă care poate fi selectată sau nu, `true` sau `false`. Cei mai utilizaţi constructori sunt:

- `public Checkbox(String eticheta) throws HeadlessException`

- `public Checkbox(String eticheta, boolean stare) throws HeadlessException`

Ambii creează un *checkbox* cu eticheta specificată. Prima variantă stabileşte starea iniţială ca fiind neselectată, iar cealaltă variantă va crea un *checkbox* selectat iniţial, dacă `stare` va fi `true`, sau neselectat, în caz contrar.

Pentru a capta acţiunile asupra câmpului *checkbox* va trebui ca acesta să aibă ataşat un ascultător de tip `ItemListener`. Acest lucru (ataşarea) se realizează prin apelul metodei `addItemListener()` din clasa `Checkbox`.

Interfaţa `ItemListener` are o singură metodă:

- `public void itemStateChanged(ItemEvent e)`

şi este apelată ori de câte ori starea *checkbox*-ului este modificată.

Clasa `Checkbox` pune la dispoziţie posibilitatea modificării/obţinerii etichetei şi respectiv a stării prin metodele:

- `public String getLabel()`
- `public boolean getState()`
- `public void setLabel(String etichetaNoua)`
- `public void setState(boolean stareNoua)`

Java pune la dispoziţie gruparea mai multor *checkbox*-uri. Pentru aceasta există clasa `CheckboxGroup`. Ea permite ca la un moment dat un singur câmp *checkbox* să fie selectat, iar celelalte nu. Selectarea unuia dintre ele duce la deselectarea celui care a fost selectat anterior. Singurele lucruri care diferă faţă de *checkbox*-uri sunt definirea (crearea) şi rezolvarea evenimentelor.

Pentru a crea un grup de *checkbox*-uri se utilizează constructorul implicit al clasei `CheckboxGroup`. Adăugarea unui *checkbox* se realizează fie prin apelarea constructorului:

- `public Checkbox(String eticheta, CheckboxGroup grup, boolean state)`

sau a metodei:

- `public setCheckboxGroup(CheckboxGroup grup)`

Obţinerea grupului din care face parte un *checkbox* se realizează în urma apelului metodei `getCheckboxGroup()`.

Exemplul 7.4.10. Crearea unui grup de *checkbox*-uri.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class ButonRadio extends Applet
    implements ItemListener {
  // declararea grupului de checkbox-uri
  private CheckboxGroup grup;
```

```
    // declararea checkbox-urilor
    private Checkbox red, orange, yellow;

    public void init() {
      // crearea unui grup de checkbox-uri
      grup = new CheckboxGroup();

      // crearea unui checkbox si adaugarea la grup
      red = new Checkbox("Red", grup, false);
      // adaugarea la applet
      add(red);
      // stabilirea ascultatorului
      red.addItemListener(this);

      orange = new Checkbox("Orange", grup, true);
      add(orange);
      orange.addItemListener(this);

      yellow = new Checkbox("Yellow", grup, false);
      add(yellow);
      yellow.addItemListener(this);
    }

    // metoda de interceptare a evenimentelor
    // legate de checkbox-uri
    public void itemStateChanged(ItemEvent e) {
      // daca s-a actionat asupra unui checkbox
      // care prezinta interes
      if ((e.getSource() == red) || (e.getSource() == orange)
       || (e.getSource() == yellow))
      // se va afisa un mesaj in bara de stare a navigatorului
      showStatus("Starea = " + red.getState() + " " +
                 orange.getState() + " " + yellow.getState());
    }
}
```

Se observă în bara de stare a navigatorului că la un moment dat un singur *checkbox* este selectat.

7.4.8. Liste de opţiuni

O listă de opţiuni este o listă de elemente text din care utilizatorul le poate alege pe cele dorite. Există două tipuri de liste de opţiuni după numărul de elemente care pot fi selectate la un moment dat: simple (se poate alege o singură opţiune) şi complexe (se pot alege mai multe opţiuni).

O listă de opţiuni simplă este asemănătoare cu un grup de *checkbox*-uri, în sensul că la un moment dat o singură opţiune este selectată, iar selectarea unei opţiuni duce la deselectarea celei selectate anterior. Deosebirea constă în faptul că o singură opţiune este vizibilă (cea selectată), iar în momentul în care utilizatorul doreşte selectarea altei opţiuni, se vor afişa şi celelalte.

O listă de opţiuni simplă se realizează prin instanţierea clasei `Choice`. Aceasta are doar un constructor fără parametri. Ascultătorii acestei componente grafice trebuie să implementeze interfaţa `ItemListener`.

Cele mai semnificative metode ale clasei `Choice` sunt explicate în tabelul de mai jos:

Prototipul metodei	Descrierea metodei
`public void` **`add`**`(String optiune)`	Adaugă opţiunea dată de `optiune` la listă.
`public void` **`addItemListener`**`(ItemListener ascultator)`	Stabileşte un ascultător pentru lista de opţiuni.
`public String` **`getItem`**`(int pozitie)`	Returnează opţiunea de pe poziţia specificată.
`public int` **`getItemCount`**`()`	Întoarce numărul de opţiuni din listă.
`public int` **`getSelectedIndex`**`()`	Returnează indexul opţiunii selectate.
`public String` **`getSelectedItem`**`()`	Returnează opţiunea selectată.
`public void` **`insert`**`(String optiune, int pozitie)`	Inserează opţiunea specificată pe poziţia indicată. Dacă există un element pe poziţia respectivă, atunci opţiunile următoare sunt deplasate cu o poziţie; altfel, elementul va fi inserat pe prima poziţie. Dacă elementului selectat i s-a schimbat indexul, atunci opţiunea nou inserată va deveni cea selectată.
`public void` **`remove`**`(int pozitie)`	Se elimină opţiunea de pe poziţia specificată, iar cele care urmează vor fi deplasate cu o poziţie. Dacă s-a eliminat opţiunea selectată, atunci prima opţiune va deveni selectată.
`public void` **`remove`**`(String optiune)`	Analog cu metoda precedentă cu deosebirea că se va elimina opţiunea specificată.
`public void` **`removeAll`**`()`	Se vor elimina toate opţiunile din listă.
`public void` **`select`**`(int pozitie)`	Se selectează opţiunea indicată prin poziţia sa.
`public void` **`select`**`(String optiune)`	Se selectează opţiunea indicată prin numele său.

Exemplul 7.4.11. Appletul va crea o listă de opţiuni, iar în bara de stare a navigatorului va fi afişată opţiunea selectată.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class ListaOptiuni extends Applet
   implements ItemListener {
   private Choice colorChoice;
   public void init() {
      // crearea unei liste de optiuni
      colorChoice = new Choice();
      // adaugarea optiunilor
      colorChoice.add("Rosu");
      colorChoice.add("Galben");
      colorChoice.add("Albastru");
      // adaugarea listei de optiuni la applet
      add(colorChoice);
      // stabilirea ascultatorului
      colorChoice.addItemListener(this);
   }

   // metoda de tratare a evenimentelor legate
   // de lista de optiuni
   public void itemStateChanged(ItemEvent e) {
      // daca lista de optiuni este sursa evenimentului
      if (e.getSource() == colorChoice) {
         String aChoice = e.getItem().toString();
         // se afiseaza optiunea selectata in bara de stare
         showStatus("Starea = " + aChoice);
      }
   }
}
```

O listă de opţiuni complexă este o listă de opţiuni din care unul sau mai multe elemente pot fi selectate. Clasa `List` pune la dispoziţie crearea unei liste de opţiuni care se poate comporta ca una simplă sau ca una complexă. Implicit este simplă.

Pe lângă cel implicit, clasa `List` mai posedă doi constructori frecvent utilizaţi:

- `public List(int randuri) throws HeadlessException`
- `public List(int randuri, boolean tip) throws HeadlessException`

Parametrul `randuri` se referă la numărul de opţiuni care vor fi vizibile la un moment dat, iar dacă `tip` are valoarea `true`, atunci lista de opţiuni va fi complexă. Implicit este simplă.

Cele mai importante metode ale clasei `List` sunt următoarele:

Prototipul metodei	Descrierea metodei
public void **add**(String optiune)	Adaugă opţiunea indicată la sfârşitul listei.
public void **add**(String optiune, int pozitie)	Adaugă opţiunea indicată pe poziţia specificată. Dacă indexul nu este valid, atunci opţiunea se adaugă la sfârşitul listei.
public void **addActionListener**(ActionListener ascultator)	Adaugă un ascultător de tip `ActionListener` pentru a trata evenimentele apărute în urma unui dublu-clic.
public void **addItemListener**(ItemListener ascultator)	Adaugă un ascultător de tip `ItemListener` pentru a trata evenimentele apărute în urma unui clic.
public void **deselect**(int index)	Deselectează opţiunea cu indexul specificat.
public String **getItem**(int index)	Returnează opţiunea care are indexul specificat.
public int **getItemCount**()	Returnează numărul de opţiuni din lista cu opţiuni.
public String[] **getItems**()	Întoarce un tablou de tip `String` cu toate opţiunile din listă.
public int **getRows**()	Returnează numărul de linii vizibile.
public int[] **getSelectedIndexes**()	Întoarce un tablou cu indecşii opţiunilor selectate.
public String[] **getSelectedItems**()	Întoarce un tablou cu opţiunile selectate.
public void **isIndexSelected**(int index)	Testează dacă opţiunea cu indexul indicat este selectată.
public void **isMultipleMode**()	Întoarce `true`, dacă se permite selectarea multiplă, şi `false`, în caz contrar.
public void **makeVisible**(int index)	Face vizibilă opţiunea cu indexul specificat.
public void **remove**(int index)	Elimină opţiunea cu indexul specificat.
public void **remove**(String optiune)	Elimină opţiunea indicată.
public void **removeAll**()	Elimină toate opţiunile din listă.
public void **select**(int index)	Selectează opţiunea cu indexul specificat.
public void **setMultipleMode**(boolean b)	Dacă `b` are valoarea `true`, atunci lista va deveni complexă, iar în caz contrar simplă.

Exemplul 7.4.12. Următorul applet conţine o listă cu patru opţiuni cu selectare multiplă. La un moment dat doar trei opţiuni sunt vizibile (va apărea şi o bară de defilare pentru vizualizarea celeilalte opţiuni). La apăsarea butonului `Gata` se vor afişa în bara de stare a navigatorului elementele selectate.

```
import java.applet.Applet;
import java.awt.Button;
import java.awt.List;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class ListaOptiuniComplexa
    extends Applet implements ActionListener {
  private List list;
  private Button gata;

  public void init() {
    // crearea unei liste de optiuni
    // cu selectare multipla
    list = new List(3, true);
    // adaugarea optiunilor
    list.add("Red");
    list.add("Yellow");
    list.add("Blue");
    list.add("Green");
    // adaugarea listei la applet
    add(list);

    // crearea unui buton
    gata = new Button("Gata");
    add(gata);
    gata.addActionListener(this);
  }

  public void actionPerformed(ActionEvent e) {
    if (e.getSource() == gata) {
      // obtinerea elementelor selectate
      String[] selectie = list.getSelectedItems();
      String sir = "";
      for (int i = 0; i < selectie.length; i++)
        sir = sir + selectie[i] + " ";
      // afisarea elementelor selectate
      // in bara de stare
      showStatus("Starea = " + sir);
    }
  }
}
```

7.4.9. Suprafeţe de desenare

Până acum am văzut aplicaţii în care desenul şi butoanele erau în aceeaşi fereastră. Problema este că butoanele ar putea acoperi desenul. Suprafeţele de desenare (pânzele, engl. *canvases*) rezolvă această problemă, definind o arie separată unde se plasează desenul. O suprafaţă de desenare este creată ca un obiect al unei clase derivate din clasa `java.awt.Canvas`. La rândul ei, clasa `Canvas` este derivată din clasa `Component`. Programatorul poate suprascrie funcţia `paint()`.

Exemplul 7.4.13. Adăugarea unei suprafeţe de desenare la un applet. Aceasta va avea culoarea gri şi un mesaj de probă.

```
import java.applet.Applet;
import java.awt.*;

public class ExempluCanvas extends Applet {
  // crearea unui canvas propriu
  private CanvasulMeu unCanvas = new CanvasulMeu();

  public void init() {
    // stabilirea culorii de fundal pentru canvas
    unCanvas.setBackground(Color.gray);
    // redimensionarea canvas-ului
    unCanvas.setSize(200, 200);
    // adaugarea canvas-ului la applet
    add(unCanvas);
  }
}

// definirea propriului canvas
class CanvasulMeu extends Canvas {

  // redefinirea functiei de desenare
  public void paint(Graphics g) {
    g.drawString("Proba de canvas!", 15, 15);
  }
}
```

Bineînţeles că se pot adăuga mai multe suprafeţe de desenare la un applet.

7.4.10. Panouri (*panels*)

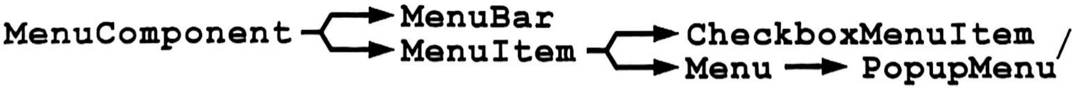
Panourile sunt utilizate pentru gruparea unui număr de componente, spre deosebire de suprafeţele grafice care sunt folosite pentru afişarea textului sau graficii. Dacă dorim un grup de butoane plasate în partea de sus a ferestrei şi alt grup în partea de jos a ferestrei, atunci putem crea două panouri pentru fiecare dintre aceste grupuri. Unui panou îi corespunde un obiect din clasa `java.awt.Panel`. Această clasă este derivată din `Container`.

Exemplul 7.4.14. Secvenţa de cod ilustrează crearea unui panou, adăugarea unor componente la panou şi adăugarea acestuia la un applet.

```
// crearea unui panou
Panel p = new Panel();
// adaugarea componentelor la panou
Button b1 = new Button("Apasa 1");
Button b2 = new Button("Apasa 2");
p.add(b1);
p.add(b2);
// adaugarea panoului la fereastra appletului
add(p);
```

7.4.11. Meniuri *pop-up*

În acest capitol vom prezenta doar meniurile *pop-up*, întrucât doar acestea pot fi utilizate de către appleturi. În aplicaţiile de sine stătătoare se utilizează meniurile din biblioteca Swing prezentată în capitolul 13.



În figura de mai sus este reprezentată ierarhia claselor utilizate pentru construirea meniurilor în AWT.

Unui meniu *pop-up* îi corespunde clasa `PopupMenu`. Aceasta are doi constructori: cel fără parametri şi unul cu un parametru de tip `String` care stabileşte un nume pentru meniu. Cea mai importantă metodă este:

- `public void show(Component origine, int x, int y)`

unde `x` şi `y` constituie coordonatele la care se va afişa meniul (colţul din stânga-sus), iar `origine` este componenta pentru care este definit spaţiul de coordonate. Această metodă duce la vizualizarea meniului.

Clasa `PopupMenu` extinde clasa `Menu`. Aceasta posedă următoarele metode importante:

Prototipul metodei	Descrierea metodei
`public MenuItem add(MenuItem menuitem)`	Adaugă elementul meniului specificat. Returnează elementul adăugat.
`public void add(String label)`	Adaugă un element de meniu cu textul specificat.
`public void insert(MenuItem menuitem, int index)`	Inserează elementul meniului pe poziţia indicată.
`public void insert(String label, int index)`	Inserează un element de meniu cu textul specificat pe poziţia indicată.
`public MenuItem getItem(int index)`	Returnează elementul care are indexul specificat.
`public int getItemCount()`	Întoarce numărul de elemente din meniu.
`public void addSeparator()`	Adaugă un separator.
`public void insertSeparator(int index)`	Inserează un separator pe poziţia indicată.
`public void remove(int index)`	Elimină elementul meniului de pe poziţia indicată.
`public void remove(MenuComponent item)`	Elimină elementul meniului specificat.
`public void removeAll()`	Elimină toate elementele din meniu.

Clasa `Menu` este derivată din `MenuItem`. Aceasta este clasa corespunzătoare unui element dintr-un meniu. Din metodele clasei `MenuItem` amintim:

Prototipul metodei	Descrierea metodei
`public String getLabel()`	Obţine eticheta elementului meniului.
`public void setLabel(String label)`	Stabileşte eticheta pentru element.
`public boolean isEnabled()`	Testează dacă elementul din meniu este disponibil.
`public void setEnabled(boolean b)`	Stabileşte dacă elementul din meniu este disponibil.

Pentru un element al meniului se poate ataşa un ascultător de tip `ActionListener` cu apelul metodei bine cunoscute `addActionListener()`.

Există un tip particular de element al meniului: de tip *checkbox*. Acesta are două stări: selectat (`true`) sau neselectat (`false`) şi este reprezentat de clasa `CheckboxMenuItem` care are următoarele metode specifice:

Prototipul metodei	Descrierea metodei
`public boolean getState()`	Obţine starea elementului.
`public void setState(boolean b)`	Setează starea elementului.

Interfaţa `ItemListener` conţine o singură metodă:

- `public void itemStateChanged(ItemEvent e)` care este invocată când un element este selectat sau deselectat.

Exemplul 7.4.15. Appletul de mai jos evidenţiază modalitatea de creare a meniurilor *pop-up* precum şi utilizarea celor două tipuri de elemente ale meniurilor:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class ExempluMeniuPopup extends Applet
    implements ActionListener, ItemListener {
  private PopupMenu meniu;
  private Button b;
  private boolean state;
  private String mesaj =
      "Acesta este un exemplu de meniu pop-up.";

  private void addOption(String opt) {
    MenuItem item = new MenuItem(opt);
     item.addActionListener(this);
    meniu.add(item);
  }

  public void init() {
    // crearea meniului
    meniu = new PopupMenu("Meniu");
    add(meniu);
    // crearea optiunilor
    addOption("Alb");
    addOption("Verde");
    addOption("Gri deschis");
    meniu.addSeparator();
    CheckboxMenuItem check = new CheckboxMenuItem("Mesaj");
    check.addItemListener(this);
    meniu.add(check);

    // crearea butonului
    b = new Button("Afisare meniu");
    b.addActionListener(this);
    add(b);

    // initializari
    state = false;
    setForeground(Color.white);
  }
```

```
    public void paint(Graphics g) {
      g.drawString(mesaj, 100, 150);
    }

    public void actionPerformed(ActionEvent e) {
      if (e.getSource() == b)
        // afisarea meniului
        meniu.show(this, 10, 10);
      else {
        // modificarea culorii fundalului
        MenuItem item = (MenuItem) e.getSource();
        String label = item.getLabel();
        if (label.equals("Verde"))
          setBackground(Color.green);
        else if (label.equals("Alb"))
          setBackground(Color.white);
        else
          setBackground(Color.lightGray);
        if (!state)
          setForeground(getBackground());
          }
    }

    public void itemStateChanged(ItemEvent e) {
      // schimbarea culorii de scriere
      setForeground(state ? getBackground(): Color.black);
      // modificarea starii
      state = ! state;
    }
  }
```

7.5. Gestionari de poziţionare (*Layout Managers*)

Un gestionar de poziţionare este un obiect al unei clase care implementează interfaţa `LayoutManager`.

Pentru stabilirea unui manager de proiecte, clasa `Container` pune la dispoziţie metoda

- `public void setLayout(LayoutManager m)`
  unde `LayoutManager` este o interfaţă implementată de toţi managerii de proiecte.

Orice container are un gestionar de poziţionare. Java pune la dispoziţie următoarele clase pentru gestionarea poziţionării:

- `BorderLayout`
- `CardLayout`
- `FlowLayout`
- `GridBagLayout`
- `GridLayout`

Pe lângă gestionarii de poziţionare corespunzători claselor amintite mai sus mai există şi gestionarul pentru poziţionare absolută. Gestionarii sunt prezentaţi separat în secţiunile următoare.

7.5.1. Gestionarul **`BorderLayout`**

Acest gestionar de poziţionare conţine cinci regiuni: nord, est, sud, vest şi centru. Fiecare regiune poate include cel mult o componentă (adăugarea unei componente implică eliminarea celei precedente). Pentru identificarea unei regiuni se utilizează atributele publice şi statice ale clasei `BorderLayout` (respectiv pentru nord, est, sud, vest şi centru):

- `public static String` **`NORTH`**
- `public static String` **`EAST`**
- `public static String` **`SOUTH`**
- `public static String` **`WEST`**
- `public static String` **`CENTER`**

Exemplul 7.5.1. Adăugarea unui buton în partea de jos a unui panou se poate realiza astfel:

```
// crearea panoului
Panel panou = new Panel();
// stabilirea gestionarului de pozitionare
panou.setLayout(new BorderLayout());
// crearea unui buton
Button butonOk = new Button("OK");
butonOk.addActionListener(this);
// pozitionarea butonului in partea de jos a panoului
add(butonOk, BorderLayout.SOUTH);
```

Dacă nu se specifică zona în care se va adăuga o componentă, atunci se va poziţiona în centru. Considerând că `panou` este obiect al clasei `Panel` şi are un gestionar de poziţionare `BorderLayout`, linia de cod:

```
panou.add(componenta);
```

este echivalentă cu:

```
panou.add(componenta, BorderLayout.CENTER);
```

O altă variantă de a adăuga o componentă într-o anumită zonă este de a specifica mai întâi regiunea sub forma unui şir de caractere şi apoi componenta.

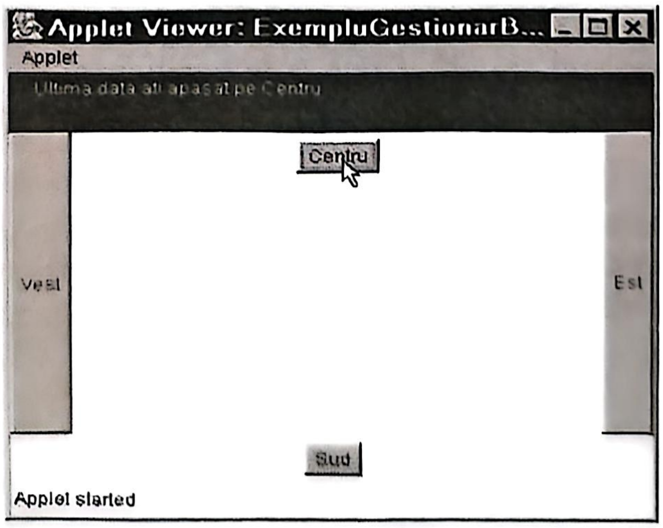
Exemplul 7.5.2. Adăugarea unor componente în cele cinci regiuni (respectiv nord, sud, vest, est şi centru) ale gestionarului `BorderLayout`.

```
add("North", componenta1);
add("South", componenta2);
add("West", componenta3);
add("East", componenta4);
add("Center", componenta5);
```

Exemplul 7.5.3. Appletul va poziţiona câte un buton în regiunile est, vest, centru şi sud, iar la acţionarea butoanelor va apărea un mesaj în regiunea nordică.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class ExempluGestionarBorderLayout
   extends Applet implements ActionListener {
  Panel bottom, centrat;
  Button sud, vest, est, centru;
  CanvasNordic nord;

  public void init() {
    // stabilirea gestionarului de pozitionare
    setLayout(new BorderLayout());

    // crearea butoanelor
    sud    = new Button("Sud");
    vest   = new Button("Vest");
    est    = new Button("Est");
    centru = new Button("Centru");

    // atasarea ascultatorului
    sud.addActionListener(this);
    vest.addActionListener(this);
    est.addActionListener(this);
    centru.addActionListener(this);

    // adaugarea butoanelor
    add("West", vest);
    add("East", est);

    // adaugarea de panouri
    bottom = new Panel();
    bottom.add(sud);
    add("South", bottom);
    centrat = new Panel();
    centrat.add(centru);
    add(centrat);

    // adaugarea canvas-ului
    nord = new CanvasNordic();
    nord.setBackground(Color.black);
    nord.setForeground(Color.white);
    nord.setSize(200, 40);
    add(nord, BorderLayout.NORTH);
  }

  // afisarea in partea nordica a unui mesaj
  public void actionPerformed(ActionEvent e) {
    Button buton = (Button) e.getSource();
    nord.mesaj = "Ultima data ati apasat pe " +
       buton.getLabel();
    nord.repaint();
  }
}

// definirea propriului canvas
class CanvasNordic extends Canvas {
  String mesaj="";
  // redefinirea functiei de desenare
  public void paint(Graphics g) {
    g.drawString(mesaj, 15, 15);
  }
}
```

În urma execuţiei appletului putem avea situaţia din figura de mai jos (când s-a apăsat pe butonul `Centru`):



Ultima data ati apasat pe Centru

Applet started

Exemplu de utilizare a gestionarului *BorderLayout*

Butoanele `Est` și `Vest` au fost adăugate direct în regiuni și de aceea au ocupat întreaga regiune. Butoanele `Centru` și `Sud` au fost incluse prin intermediul panourilor (panourile au fost incluse în regiuni, iar butoanele în panouri) și din acest motiv apar cu dimensiunile standard.

7.5.2. Gestionarul **`CardLayout`**

Acest gestionar permite afișarea unei singure componente la un moment dat. De obicei, se adaugă panouri care conțin componentele dorite, iar pentru fiecare panou putem să stabilim gestionarul de poziționare dorit.

Clasa `CardLayout` are doi constructori: unul fără parametri și unul cu doi parametri de tip `int` ce indică marginea care este lăsată la stânga și la dreapta, respectiv sus și jos:

- `public CardLayout(int orizontal, int vertical)`

Cele mai semnificative metode din clasa `CardLayout` sunt:

Prototipul metodei	Descrierea metodei
`void` **`addLayoutComponent`**`(Component comp, Object constrangeri)`	Adaugă componenta specificată și stabilește pentru aceasta un nume (dat de parametrul `constrangeri` care trebuie să fie de tip `String`).
`void` **`first`**`(Container parinte)`	Se afișează prima componentă.
`void` **`last`**`(Container parinte)`	Se afișează ultima componentă.
`void` **`next`**`(Container parinte)`	Se afișează componentă următoare.
`void` **`previous`**`(Container parinte)`	Se afișează componentă precedentă.
`boolean` **`show`**`(Container parinte, String nume)`	Se activează componenta care are atașat numele specificat.

Menționăm că parametrul `parinte` este referința la obiectul care are stabilit gestionarul `CardLayout`.

Exemplul 7.5.4. Appletul are un gestionar de poziționare de tip `CardLayout` și conține trei componente (panouri mai exact). La apăsarea butoanelor se trece de la o componentă la alta.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class ExempluGestionarCardLayout
   extends Applet implements ActionListener {

  Button next1, next2, next3;
```

```
Panel p1, p2, p3;
TextArea t1, t2, t3;
CardLayout gestionar;
int pagina = 1;

public void init() {
  gestionar = new CardLayout();
  setLayout(gestionar);

  // crearea butoanelor
  next1 = new Button("Mai departe...");
  next2 = new Button("Mai departe...");
  next3 = new Button("La inceput");
  next1.addActionListener(this);
  next2.addActionListener(this);
  next3.addActionListener(this);

  // crearea panourilor
  p1 = new Panel();
  p2 = new Panel();
  p3 = new Panel();

  // adaugarea panourilor
  add(p1, "P1");
  add(p2, "P2");
  add(p3, "P3");

  // completarea primului panou
  t1 = new TextArea("Aceasta este prima pagina.\n"+
    "Apasati butonul din dreapta.",
    10,40,TextArea.SCROLLBARS_NONE);
  t1.setEditable(false);
  p1.add(t1);
  p1.add(next1);

  // completarea celui de-al doilea panou
  t2 = new TextArea("Ati ajuns la pagina a doua.\n" +
    "Pentru a merge mai departe apasati butonul alaturat",
    10,30,TextArea.SCROLLBARS_NONE);
  t2.setEditable(false);
  p2.add(t2);
  p2.add(next2);

  // completarea celui de-al treilea panou
  p3.add(next3);
```

```
    }

    public void actionPerformed(ActionEvent e) {
      gestionar.next(this);
    }
}
```

7.5.3. Gestionarul `FlowLayout`

Gestionarul de poziționare `FlowLayout` este implicit pentru containerele `Panel` și `Applet`. Acesta conține o singură regiune în care componentele sunt aliniate centrat și sunt plasate de la stânga la dreapta. În cazul în care sunt prea mult componente pe un singur rând, atunci fie se redimensionează containerul-părinte, fie se trece la rândul următor.

La rularea programelor din exemplele 7.5.3 și 7.5.4 se poate observa modul de adăugare a componentelor într-un container care are un gestionar `FlowLayout`.

Implicit, componentele sunt aliniate centrat. Metoda `setAlignment()` stabilește alinierea rândurilor cu componentele. Aceasta are un singur parametru întreg și poate lua următoarele valori predefinite: `FlowLayout.LEFT` (pentru aliniere la stânga), `FlowLayout.RIGHT` (pentru aliniere la dreapta) și `FlowLayout.CENTER` (pentru aliniere centrată). Componentele se vor adăuga de la stânga la dreapta, indiferent de tipul alinierii.

Exemplul 7.5.5. Appletul va afișa o etichetă aliniată la dreapta.

```
import java.applet.Applet;
import java.awt.*;

public class ExempluGestionarFlowLayout
   extends Applet {

  private Label l;

  public void init() {
    // crearea si adaugarea  unei etichete
    l = new Label("Aliniere la dreapta.");
    add(l);
    // obtinerea gestionarului
    FlowLayout gestionar = (FlowLayout) getLayout();
    // alinierea la dreapta
    gestionar.setAlignment(FlowLayout.RIGHT);
  }
}
```

7.5.4. Gestionarul `GridLayout`

Gestionarul de poziţionare `GridLayout` conţine o grilă de celule, toate cu aceleaşi dimensiuni. Fiecare componentă este adăugată într-o celulă de la stânga la dreapta şi de sus în jos. Această ordine poate fi schimbată prin apelul metodei `setComponentOrientation()` din clasa `Component` (mai exact, al obiectului care are stabilit acest gestionar de poziţionare).

Constructorii clasei `GridLayout` (în afară de cel fără parametri) sunt:

- `public GridLayout(int nrRanduri, int nrColoane)`
- `public GridLayout(int nrRanduri, int nrColoane, int orizontal, int vertical)`

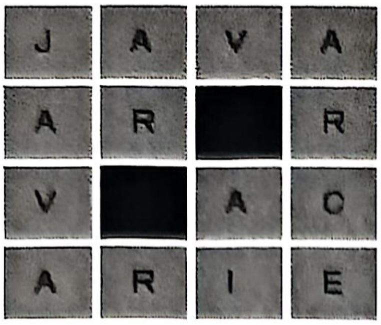
unde primii doi parametri se referă la numărul de rânduri şi coloane al grilei, iar `oriz` şi `vert`, la distanţa dintre celule pe orizontală, respectiv pe verticală.

Dacă numărul de componente depăşeşte dimensiunea grilei, aceasta se va redimensiona corespunzător, pentru a putea încăpea toate elementele sale.

Exemplul 7.5.6. Appletul afişează rezolvarea unui rebus de 4 linii şi 4 coloane.

```
import java.applet.Applet;
import java.awt.*;

public class ExempluGestionarGridLayout extends Applet {
  public Label rebus[][] = new Label[4][4];

  public void init() {
    // stabilirea gestionarului GridLayout
    setLayout(new GridLayout(4,4,3,3));
    // crearea etichetelor
    int i,j;
    String[] litere = {
      "J", "A", "V", "A",
      "A", "R", " ", "R",
      "V", " ", "A", "C",
      "A", "R", "I", "E" };
    for(i = 0;i < 4;i++)
      for(j = 0;j < 4;j++) {
        rebus[i][j] = new Label();
        rebus[i][j].setText("    " + litere[4 * i + j]);
        rebus[i][j].setBackground(Color.lightGray);
        add(rebus[i][j]);
      }
    // colorarea patratelor neutilizate
    rebus[2][1].setBackground(Color.black);
    rebus[1][2].setBackground(Color.black);
    // stabilirea dimensiunii appletului
    setSize(100,130);
  }
}
```

La execuția appletului va apărea:



Exemplu de utilizare a gestionarului *GridLayout*

7.5.5. Gestionarul `GridBagLayout`

Gestionarul `GridBagLayout` este unul mai flexibil și permite alinierea verticală și orizontală a componentelor pe baza unor constrângeri. Gestionarul este asemănător cu `GridLayout` cu deosebirea că o componentă se poate întinde pe mai multe celule. Clasa care permite specificarea constrângerilor este `GridBagConstraints`. Setarea și respectiv obținerea constrângerilor pentru o componentă se realizează cu ajutorul următoarelor metode din clasa `GridBagLayout`:

- `public void setConstraints(Component comp, GridBagConstraints c);`
- `public GridBagConstraints getConstraints(Component comp);`

unde `comp` este componenta în cauză, iar `c` specifică constrângerile ce vor fi aplicate componentei.

Cele mai importante atribute din clasa `GridBagConstraints`, utile pentru stabilirea poziției componentei în cadrul grilei, sunt prezentate în tabelul de mai jos.

Prototipul metodei	Descrierea metodei
`int` **`gridheight`**	Indică numărul de celule pe verticală pe care îl ocupă componenta.
`int` **`gridwidth`**	Indică numărul de celule pe orizontală pe care îl ocupă componenta.
`int` **`gridx`**	Indică rândul pe care va fi situată componenta.
`int` **`gridy`**	Indică pe care coloană va fi situată componenta.

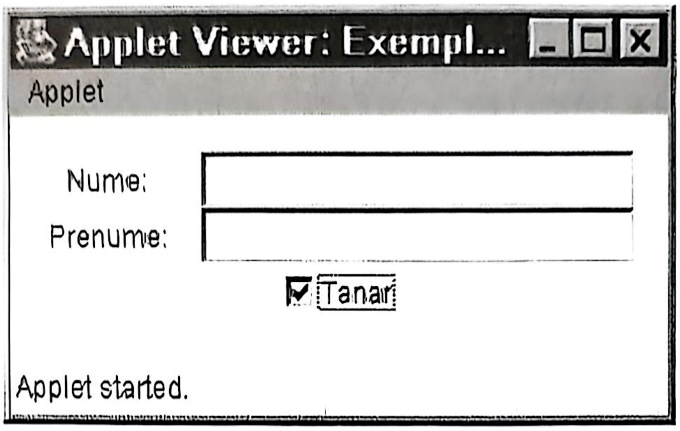
Următorul exemplu prezintă modalitatea de a utiliza gestionarul `GridBagLayout`, precum și stabilirea constrângerilor pentru o cât mai bună aliniere a componentelor.

Exemplul 7.5.7. Appletul utilizează gestionarul `GridBagLayout`.

```
import java.applet.Applet;
import java.awt.*;

public class ExempluGestionarGridBagLayout extends Applet {
  public void init() {
    // stabilirea gestionarului de pozitionare

    GridBagLayout grid = new GridBagLayout();
    setLayout(grid);

    // crearea componentelor
    Label l1 = new Label("Nume: ");
    Label l2 = new Label("Prenume: ");
    TextField t1 = new TextField(25);
    TextField t2 = new TextField(25);
    Checkbox cb = new Checkbox("Tanar", true);

    // stabilirea constrangerilor
    GridBagConstraints c = new GridBagConstraints();
    // se stabileste prima coloana
    c.gridx = 0;
    // se stabileste primul rand
    c.gridy = 0;
    // componenta va ocupa 2 celule pe orizontala
    c.gridwidth = 2;
    // stabilirea constrangerilor pentru l1
    grid.setConstraints(l1,c);
    // se stabileste al doilea rand
    c.gridy = 1;
    // stabilirea constrangerilor pentru l2
    grid.setConstraints(l2,c);
    // se va stabili a treia coloana
    c.gridx = 2;
    grid.setConstraints(t2,c);
    // se stabileste primul rand
    c.gridy = 0;
    grid.setConstraints(t1,c);
    // se va stabili coloana a doua, randul 3
    c.gridx = 1;
    c.gridy = 2;
    // se vor ocupa 3 celule pe orizontala
    c.gridwidth = 3;
    grid.setConstraints(cb,c);
```

```
        // adaugarea componentelor
        add(cb);
        add(l1);
        add(l2);
        add(t1);
        add(t2);

        // stabilirea dimensiunii appletului
        setSize(300,100);
    }
}
```

Appletul va afişa următoarele componente (care nu au efect, întrucât nu le-au fost ataşaţi ascultători):



Exemplu de utilizare a gestionarului *GridBagLayout*

7.5.6. Poziţionarea absolută

Pentru a plasa componentele fără a utiliza un gestionar predefinit vom apela metoda `setLayout()` cu valoarea `null`:

- `setLayout(null);`

Pentru dimensionarea şi plasarea componentelor putem utiliza metoda:

- `void setBounds(int x, int y, int latime, int inaltime);`

unde x şi y reprezintă coordonatele poziţiei componentei în container, iar parametrii `latime` şi `inaltime` reprezintă dimensiunile componentei. Acelaşi lucru se poate realiza şi cu metodele `setLocation()` şi `setSize()`:

- `void setLocation(int x, int y);`
- `void setSize(int latime, int inaltime);`

unde parametrii au aceeaşi semnificaţie ca mai sus.

Exemplul 7.5.8. Appletul afişează trei butoane aşezate în diagonală.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
```

```
public class PozitionareAbsoluta extends Applet

    implements ActionListener {

   String sir = "";
   String butonSelectat;
   Button b1, b2, b3;

   public void init() {
      // stabilirea pozitionarii absolute
      setLayout(null);

      // crearea de butoane
      b1 = new Button("Buton1");
      b2 = new Button("Buton2");
      b3 = new Button("Buton3");
      // adaugarea la applet
      add(b1);
      add(b2);
      add(b3);
      // stabilirea dimensiunilor si a pozitiei
      b1.setBounds(  0,  0, 60, 30);
      b2.setBounds( 60, 30, 60, 30);
      b3.setBounds(120, 60, 60, 30);
      // stabilirea ascultatorilor
      b1.addActionListener(this);
      b2.addActionListener(this);
      b3.addActionListener(this);
   }

   public void actionPerformed(ActionEvent ev) {
      Button b = (Button) ev.getSource();
      sir = "Ati apasat " + b.getLabel();
      repaint();
   }

   public void paint(Graphics g) {
      g.drawString(sir, 10, 120);
   }
}
```

7.6. Ferestre

Ferestrele în AWT au la bază clasa `java.awt.Window`. Aceasta este derivată din `Container` şi nu conţine implicit margini sau vreo bară de meniuri sau unelte. Această clasă nu este utilizată, de regulă, direct pentru construirea ferestrelor. Sunt utilizate clasele derivate, cum ar fi `Dialog` (pentru ferestre de dialog) sau `Frame` (pentru fereastra unei aplicaţii).

7.6.1. Clasa `java.awt.Window`

Este de remarcat faptul că ferestrele formează o ierarhie. Fiecare fereastră are o fereastră-părinte şi posedă (sau nu) mai multe subferestre.

Cele mai importante metode publice din clasa `Window` sunt următoarele:

Prototipul metodei	Descrierea metodei
`void` **`addWindowListener`**`()`	Stabileşte un ascultător pentru evenimentele legate de fereastră.
`void` **`dispose`**`()`	Eliberează resursele native ale ferestrei şi implică dispariţia acesteia de pe ecran.
`Component` **`getFocusOwner`**`()`	Returnează componenta activă din fereastră sau `null`, dacă nici o componentă nu este activă.
`Window[]` **`getOwnedWindows`**`()`	Returnează un vector cu subferestrele ferestrei curente.
`Window` **`getOwner`**`()`	Întoarce părintele acestei ferestre.
`void` **`setVisible`**`(boolean b)`	Afişează fereastra cu toate componentele sale pentru valoarea `true` a parametrului `b`, iar în caz contrar conduce la ascunderea ferestrei.
`boolean` **`isActive`**`()`	Testează dacă fereastra este activă (pentru `Frame` sau `Dialog`).
`boolean` **`isFocused`**`()`	Testează dacă fereastra are focusul.
`boolean` **`isShowing`**`()`	Testează dacă fereastra este vizibilă pe ecran.
`void` **`pack`**`()`	Fereastra şi componentele sale vor avea dimensiunile optime.
`void` **`toBack`**`()`	Trimite fereastra în fundal şi aceasta devine inactivă.
`void` **`toFront`**`()`	Trimite fereastra în prim-plan şi aceasta devine activă.

Aceste metode se vor regăsi şi la clasele derivate, cu aceleaşi funcţionalităţi.

Interfaţa `WindowListener` defineşte şapte metode. Acestea sunt:

Prototipul metodei	Descrierea metodei
`void` **`windowActivated`** `(WindowEvent e)`	Este invocată când fereastra este activată.
`void` **`windowClosed`** `(WindowEvent e)`	Este invocată după execuţia metodei `dispose()`.
`void` **`windowClosing`** `(WindowEvent e)`	Este invocată când fereastra se doreşte a fi inchisă de către utilizator.
`void` **`windowDeactivated`** `(WindowEvent e)`	Este invocată când fereastra nu mai este activată.
`void` **`windowDeiconified`** `(WindowEvent e)`	Este invocată când fereastra trece din starea minimizată în starea normală.
`void` **`windowIconified`** `(WindowEvent e)`	Este invocată când fereastra trece din starea normală în starea minimizată.
`void` **`windowOpened`** `(WindowEvent e)`	Este invocată la prima afişare pe ecran a ferestrei.

Pentru cazul în care programatorul doreşte să definească o clasă care să implementeze această interfaţă, există şi posibilitatea extinderii clasei `WindowAdapter`. Aceasta implementează interfaţa în cauză şi implicit nu are nici un efect. Pentru a da anumite funcţionalităţi programului în urma apariţiei anumitor evenimente, vom redefini metodele dorite. De exemplu, dacă dorim doar să facem posibilă închiderea ferestrei, vom redefini numai metoda `windowClosing()`. Avantajul constă în faptul că nu trebuie să mai implementăm toate metodele.

Exemplul 7.6.1. **Implementarea interfeţei `WindowListener` prin intermediul clasei `WindowAdapter`:**

```
import java.awt.event.*;

public class FereastraAscultator extends WindowAdapter {
  public void windowClosing(WindowEvent e) {
    e.getWindow().dispose();
  }
}
```

7.6.2. Clasa `java.awt.Frame`

Clasa `Frame` corespunde ferestrelor aplicaţiilor. Aceasta este derivată din clasa `Window` şi moşteneşte metodele sale, o parte dintre ele fiind descrise în secţiunea precedentă.

Cei mai utilizaţi constructori sunt cel fără parametri şi cel cu un parametru de tip `String` care stabileşte titlul ferestrei (cel care apare în bara ferestrei). Cele mai importante metode publice din clasa `Frame` sunt următoarele:

Prototipul metodei	Descrierea metodei
`static Frame[] getFrames()`	Returnează un vector cu toate obiectele de clasă `Frame` create de aplicaţie.
`MenuBar getMenuBar()`	Întoarce meniul de tip bară al aplicaţiei.
`void setMenuBar(MenuBar meniuBara)`	Stabileşte meniul de tip bară al aplicaţiei.
`String getTitle()`	Returnează titlul ferestrei.
`void setTitle(String titlu)`	Schimbă titlul ferestrei.
`boolean isResizable()`	Indică dacă fereastra se poate redimensiona.
`void setResizable(boolean b)`	Stabileşte dacă fereastra poate fi redimensionată.

O aplicaţie care conţine o fereastră principală va avea de cele mai multe ori clasa principală derivată din clasa `Frame`. Gestionarul de poziţionare implicit este `BorderLayout`.

Exemplul 7.6.2. O aplicaţie simplă de sine stătătoare:

```
import java.awt.*;

public class AplicatieSimpla extends Frame {
  public AplicatieSimpla() {
    // apelarea constructorului superclasei
    super("AplicatieSimpla");
    // stabilirea ascultatorului pentru fereastra
    addWindowListener(new FereastraAscultator());
    // stabilirea pozitiei pe ecran si a dimensiunii
    setBounds(100,50,200,100);
    // crearea si adaugarea unei etichete
    Label eticheta = new Label(
       "Prima aplicatie de sine statatoare.");
    add(eticheta);
  }

  public static void main(String[] args) {
    AplicatieSimpla app = new AplicatieSimpla();
    // afisarea ferestrei
    app.setVisible(true);
  }
}
```

Clasa `FereastraAscultator` este cea definită în exemplul 7.6.1.

7.6.3. Clasa `java.awt.Dialog`

Clasa `Dialog` permite lucrul cu ferestre de dialog. Acestea nu pot fi maximizate sau minimizate. Gestionarul implicit de poziţionare este tot `BorderLayout`. Clasa `Dialog` posedă următorii constructori importanţi:

- `public Dialog(Dialog d)`
- `public Dialog(Dialog d, String titlu)`
- `public Dialog(Dialog d, String titlu, boolean modal)`
- `public Dialog(Frame f)`
- `public Dialog(Frame f, String titlu)`
- `public Dialog(Frame f, String titlu, boolean modal)`

unde `f` şi `d` sunt ferestrele-părinte, `titlu` este titlul ferestrei, iar `modal` indică dacă fereastra de dialog este modală sau nu. O fereastră este *modală*, dacă odată activată nu se va putea reactiva fereastra-părinte decât după închiderea acesteia. O fereastră este *nemodală*, dacă permite activarea ferestrei-părinte fără ca aceasta să se închidă.

Clasa `Dialog` este derivată din `Window` şi are în plus două metode importante care se referă la tipul ferestrei (modală sau nemodală):

- `public boolean isModal()`
- `public void setModal(boolean b)`

Prima metodă testează dacă fereastra este modală, iar a doua setează caracterul modal (dacă `b` are valoarea `true`).

Exemplul 7.6.3. Aplicaţia creează o fereastră de dialog şi o activează la apăsarea unui buton. Fereastra de dialog se „ascunde" la apăsarea butonului `Ok` sau la închiderea acesteia.

```
import java.awt.*;
import java.awt.event.*;

public class ExempluDialog extends Frame
   implements ActionListener {
   Dialog dialog;

public ExempluDialog() {
    // apel la constructorul superclasei
    super("Exemplu de Dialog");
    // stabilirea pozitiei si a dimensiunilor
    setBounds(100,200,250,150);
    // adaugarea unui ascultator pentru fereastra
    addWindowListener(new FereastraAscultator());
    // crearea unui buton
    Button b = new Button("Dialog");
    add("North",b);
    b.addActionListener(this);
    // crearea unei etichete
    Label l = new Label(
       "Apasati pe butonul Dialog!");
    add(l);
    // crearea ferestrei de dialog
    dialog = new Dialog(this, "Test");
    FereastraDialogAscultator ascultator =
       new FereastraDialogAscultator(dialog);
    dialog.addWindowListener(ascultator);
    dialog.setBounds(150,250,150,100);
    // crearea unui buton si adaugarea la dialog
    Button ok = new Button("Ok");
    ok.addActionListener(ascultator);
    dialog.add(ok);
  }

  public static void main(String[] args) {
    ExempluDialog app = new ExempluDialog();
    app.setVisible(true);
  }

  public void actionPerformed(ActionEvent e) {
    dialog.setVisible(true);
  }
}

class FereastraDialogAscultator
   extends WindowAdapter
   implements ActionListener {

  private Dialog dialog;

  public FereastraDialogAscultator(Dialog dialog) {
    this.dialog = dialog;
  }

  public void windowClosing(WindowEvent e) {
    dialog.setVisible(false);;
}

  public void actionPerformed(ActionEvent e) {
    dialog.setVisible(false);
  }
}
```

7.6.4. Clasa `java.awt.FileDialog`

Selectarea numelor subdirectoarelor şi a numelor fişierelor se realizează cu o fereastră de dialog specială pentru fişiere. Acesteia îi corespunde clasa `FileDialog` care este derivată din `Dialog`. Modul exact de apariţie depinde de sistemul de operare.

Există şase constructori pentru clasa `FileDialog`:

- `public FileDialog(Frame f)`
- `public FileDialog(Frame f, String titlu)`
- `public FileDialog(Frame f, String titlu, int mod)`
- `public FileDialog(Dialog d)`
- `public FileDialog(Dialog d, String titlu)`
- `public FileDialog(Dialog d, String titlu, int mod)`

unde parametrii au aceeaşi semnificaţie ca la constructorii clasei `Dialog`, care au fost prezentaţi în subcapitolul 7.6.3, excepţie făcând parametrul `mod`, care indică dacă fereastra este utilizată pentru salvarea sau încărcarea fişierului (sau directorului). Pentru acest parametru clasa `FileDialog` pune la dispoziţie doi membri publici şi statici de tip `int`: `SAVE` pentru salvare şi `LOAD` pentru încărcare.

Metodele publice cele mai semnificative din clasa `FileDialog` sunt următoarele:

Prototipul metodei	Descrierea metodei
`String` **`getDirectory`**`()`	Obţine directorul selectat.
`String` **`getFile`**`()`	Obţine un nume pentru fişierul selectat.
`void` **`setFile`**`(String fisier)`	Stabileşte un nume de fişier implicit.
`int` **`getMode`**`()`	Returnează tipul ferestrei: de încărcare sau de salvare.
`void` **`setMode`**`(int mode)`	Stabileşte dacă fereastra este pentru încărcare sau pentru salvare.
`FilenameFilter` **`getFilenameFilter`**`()`	Returnează filtrul stabilit pentru fişiere.
`void` **`setFilenameFilter`**`(FilenameFilter filtru)`	Stabileşte un filtru pentru fişierele ce vor apărea în fereastră.

Pentru a stabili un filtru trebuie să construim o clasă care să implementeze interfaţa `FilenameFilter`. Aceasta face parte din pachetul `java.io` şi conţine o singură metodă care testează dacă un anumit fişier dintr-un director specificat este acceptat sau nu:

- `public boolean accept(File dir, String fisier)`

Exemplul 7.6.4. Caseta de dialog pentru fişiere apare când utilizatorul apasă butonul `Load`, apoi numele fişierului selectat este scris într-un câmp text.

```
import java.awt.*;
import java.awt.event.*;
```

```
public class ExempluFileDialog extends Frame
   implements ActionListener {

  private Button butonIncarca;
  private FileDialog obtineFisier;
  private TextField campText;

  public ExempluFileDialog() {
    super("Exemplu de FileDialog");
    setLayout(new FlowLayout());
    setSize(645, 382);
    addWindowListener(new FereastraAscultator());

    butonIncarca = new Button("Load");
    add(butonIncarca);
    butonIncarca.addActionListener(this);

    campText = new TextField(20);
    add(campText);
  }

  public static void main(String[] args) {
    ExempluFileDialog app = new ExempluFileDialog();
    app.setVisible(true);
  }

  public void actionPerformed(ActionEvent event) {
    String numeFisier;
    // crearea ferestrei de dialog pentru incarcarea fisierului
    obtineFisier = new FileDialog(this, "Obtine fisier",
        FileDialog.LOAD);
    // activarea ferestrei
    obtineFisier.setVisible(true);
    // obtinerea fisierului
    numeFisier = obtineFisier.getFile();
    // afisarea in campul text
    campText.setText(numeFisier);
  }
}
```

7.7. Managerul de securitate

O aplicaţie Java care accesează resursele sistem, cum ar fi monitorul, sistemul de fişiere, firele de execuţie, procesele şi reţeaua, poate fi controlată dintr-un singur punct cu un manager de securitate. Clasa care implementează această funcţionalitate în Java este `java.lang.SecurityManager`.

Platforma Java furnizează un mecanism implicit de securitate care este suficient pentru cele mai multe aplicaţii. Însă pentru aplicaţiile Java care necesită încărcare explicită de clase este nevoie să scriem propriul manager de securitate.

O instanţă a clasei `SecurityManager` poate fi creată doar o dată în mediul de execuţie Java. Apoi, orice acces către o resursă sistem fundamentală este filtrată prin metode specifice ale obiectului de tip `SecurityManager`. Prin intermediul acestui obiect specializat putem implementa politici de securitate simple sau complexe pentru accesul la resurse individuale.

Când începe execuţia unui program Java, se aşteaptă instalarea unui obiect `SecurityManager`. Dacă nu este instalat nici un manager de securitate (se utilizează `null`), atunci este permis orice tip de acces, aşa încât sistemul de execuţie Java va executa orice activitate la acelaşi nivel de acces ca şi alte programe care rulează cu autoritate de utilizator. Dacă aplicaţia care rulează necesită un mediu sigur, atunci se poate instala un obiect `SecurityManager` apelând metoda `System.setSecurityManager()`. De exemplu, unele navigatoare Web instalează un obiect `SecurityManager` înaintea rulării oricărui applet Java.

Aplicaţiile care doresc instalarea unui obiect `SecurityManager` trebuie să deriveze clasa abstractă `java.lang.SecurityManager`. Deşi această clasă este abstractă, ea nu conţine nici o metodă abstractă, singurul motiv fiind că implementarea implicită nu este foarte utilă. Implicit, fiecare metodă de securitate din clasa `SecurityManager` este implementată pentru a furniza cel mai strict nivel de securitate, adică un obiect `SecurityManager` va respinge implicit toate cererile.

Exemplul 7.7.1. Următorul program Java instalează cel mai restrictiv obiect `SecurityManager`, astfel:

```
class TataRau extends SecurityManager {}

public class AplicatiaUnu {
   public static void main(String args[]) {
      System.setSecurityManager(new TataRau());
      // nu avem nici un fel de acces la fisiere, retea,
      // ferestre etc.
      System.out.println("Afisam un text sigur!");
   }
}
```

Programul `AplicatiaUnu` de mai sus nu poate decât să citească din `System.in` şi să scrie în `System.out`. Orice încercare de a citi sau scrie alte fişiere, de a accesa reţeaua sau chiar a deschide o fereastră va implica aruncarea unei excepţii `SecurityException`.

În exemplul 7.7.1, s-a instalat un obiect `SecurityManager` prea dur (intitulat, de altfel, `TataRau`). Odată instalat, este imposibil să schimbăm obiectul `SecurityManager` în vreun mod, deoarece securitatea acestuia nu este dependentă de obiectul `SecurityManager` însuşi, ci este construit deja în mediul de execuţie Java.

Astfel, pentru a avea un obiect `SecurityManager` folositor va trebui să suprascriem metodele care sunt consultate pentru diferitele tipuri de acces ale unor resurse. Acestea nu se apelează, de obicei, de programator (deşi se poate), ci de nucleul Java înaintea acordării permisiunii unui tip particular de acces. Iată câteva dintre cele mai importante metode publice ale clasei `SecurityManager`:

Prototipul metodei	Descrierea metodei
`void` **`checkAccess`**`(Thread fir)`	Verifică accesul la firul de execuţie `fir`.
`void` **`checkExit`**`(int stare)`	Pentru execuţia `System.exit(stare);`.
`void` **`checkExec`**`(String comanda)`	Verifică execuţia procesului `comanda`.
`int` **`checkRead`**`(String fisier)`	Verifică drepturile de citire din fişierul specificat.
`void` **`checkWrite`**`(String fisier)`	Verifică drepturile de scriere în fişierul specificat.
`void` **`checkDeleted`**`()`	Verifică drepturile de ştergere din fişierul specificat.
`void` **`checkConnect`**`(String gazda, int port)`	Verifică accesul unui socket la serverul `gazda` şi la portul `port`.
`void` **`checkListen`**`(String gazda, int port)`	Verifică dreptul de a crea un socket la portul precizat.
`void` **`checkAccept`**`(String gazda, int port)`	Verifică dreptul de acceptare a unei conexiuni la `gazda` şi adresa `port`.
`void` **`checkPropertyAccess`**`(String cheie)`	Verifică accesul la proprietăţile sistem.
`boolean` **`checkTopLevelWindow`**`(Object fereastra)`	Verifică dreptul de aducere în prim-plan a ferestrei specificate.

Dacă accesul nu este permis, acestea aruncă o excepţie `SecurityException`. Din metodele de mai sus, `checkTopLevelWindow()` este diferită prin faptul că întoarce `true`, indicând astfel permisiunea accesului. Chiar dacă întoarce `false`, accesul este permis, însă se va afişa un chenar care identifică o fereastră nesigură (de neîncredere). Unele metode aruncă excepţia `NullPointerException` în cazul în care este transmis ca parametru valoarea `null`.

Exemplul 7.7.2. Vom implementa un manager de securitate care va permite deschiderea în acces de citire numai a fişierelor care au extensia `.java`:

```
class ExtensieJava extends SecurityManager {
   public void checkRead(String s) {
      if (s.endsWith(".java")) return ;
      else throw new SecurityException(
         "Accesul la citire este permis doar fisierelor" +
         "cu extensia .java");
   }
}
```

Odată ce managerul `ExtensieJava` este instalat, orice încercare de a citi alte fişiere decât cele cu extensia `.java` va arunca o excepţie `SecurityException` însoţită de mesajul din program. Toate celelalte metode de securitate sunt moştenite din clasa `SecurityManager`, deci vor fi restrictive.

Toate restricţiile pentru appleturi dintr-o aplicaţie care vizualizează appleturi (engl. *appletviewer*) sunt verificate de managerul de securitate, inclusiv dacă i se permite unui cod nesigur (de neîncredere) să fie încărcat prin reţea în vederea execuţiei acestuia.

7.8. Concluzii

Acest capitol a debutat cu prezentarea appleturilor, cu specificarea modului de includere a acestora în paginile Web, transmiterea parametrilor, modul de funcţionare, comunicarea dintre appleturile care sunt în acelaşi document Web, redarea clipurilor şi a imaginilor. De altfel, s-a prezentat şi modalitatea de creare a unei aplicaţii care să fie simultan applet şi aplicaţie de sine stătătoare.

S-au prezentat apoi aplicaţiile conduse de evenimente, modalitatea de desenare, utilizarea componentelor şi a containerelor, a elementelor de formular pentru introducerea facilă a datelor, tratarea evenimentelor prin intermediul ascultătorilor şi adaptorilor. De asemenea, s-au prezentat gestionarii de poziţionare a subcomponentelor unui container, precum crearea de ferestre şi de aplicaţii de sine stătătoare prin utilizarea bibliotecii AWT.

S-a descris în final modalitatea de stabilire a unei politici de restricţii pentru aplicaţii prin intermediul managerului de securitate.

7.9. Test-grilă

Întrebarea 7.9.1. Care dintre afirmaţiile de mai jos referitoare la appleturi sunt adevărate?

a) Sunt executate de către serverul Web.
b) Sunt stocate pe serverul Web.
c) Sunt compilate de navigatorul Web.
d) Sunt executate de navigatorul Web.

Întrebarea 7.9.2. Precizaţi care dintre afirmaţiile următoare sunt false:

a) Orice navigator Web poate rula appleturi.
b) Appleturile pot fi stocate pe orice server Web.
c) Appleturile pot fi stocate local, pe calculatorul clientului.
d) Un document Web poate conţine cel mult un applet.

Întrebarea 7.9.3. Un applet poate fi:

a) executat cu `appletviewer` fără să creăm o pagină HTML.
b) simultan şi aplicaţie de sine stătătoare.
c) executat cu ajutorul comenzii `java`.
d) compilat doar la încărcarea acestuia în navigatorul Web.

Întrebarea 7.9.4. Care dintre propoziţiile următoare referitoare la metodele unui applet sunt adevărate?

a) Trebuie să redefinim măcar o metodă, altfel obţinem eroare la compilare.
b) Sunt apelate automat de navigatorul Web.
c) Pot fi apelate direct de către utilizatori.
d) Nu se pot declara noi metode.

Întrebarea 7.9.5. Care dintre afirmaţiile următoare sunt false?

a) Doar appleturile din aceeaşi pagină Web pot comunica între ele.
b) Un applet poate fi configurat din cadrul paginii Web.
c) Un document Web poate conţine un applet de pe alt server Web.
d) În anumite situaţii un applet poate avea acces la sistemul de fişiere local.

Întrebarea 7.9.6. Un applet poate conţine:

a) imagini stocate în fişiere externe;
b) elemente de redare a sunetelor aflate în fişiere externe;
c) texte de diferite dimensiuni şi culori;
d) figuri geometrice bidimensionale;
e) Nu poate conţine simultan imagini, desene, text şi audio.

Întrebarea 7.9.7. O componentă grafică trebuie neapărat să:

a) conţină o reprezentare grafică;
b) aibă ataşat un ascultător pentru interceptarea acţiunilor;
c) fie vizibilă;
d) fie ataşată unei suprafeţe de desenare.

Întrebarea 7.9.8. Care dintre următoarele clase nu reprezintă componente grafice?

a) `Panel`
b) `Canvas`
c) `SoundComponent`
d) `Container`

e) `Applet`
f) `List`

Întrebarea 7.9.9. Care dintre gestionarii de poziţionare de mai jos pot fi utilizaţi de appleturi?

a) `GridBagLayout`
b) `TableLayout`
c) `DefaultLayout`
d) `FlowLayout`
e) `WindowLayout`

Întrebarea 7.9.10. Care dintre următoarele metode trebuie implementate pentru tratarea evenimentului de apăsare a unui buton?

a) Nu trebuie definită nici o metodă.
b) `actionPerformed()`
c) `actionListener()`
d) `itemStateChanged()`

Întrebarea 7.9.11. Care dintre următoarele clase definesc ferestre?

a) `FileDialog`
b) `WindowListener`
c) `Applet`
d) `Frame`
e) `Applet`
f) `MainWindow`

Întrebarea 7.9.12. Un manager de securitate poate stabili:

a) drepturile de acces la fişiere;
b) restricţii pentru executarea anumitor aplicaţii;
c) ce appleturi au permisiunea de a accesa sistemul local de fişiere;
d) doar restricţii, nu şi drepturi.

Întrebarea 7.9.13. Ce metodă trebuie definită pentru a putea desena pe suprafaţa appletului?

a) Nu trebuie definită nici o metodă.
b) `update()`
c) `paint()`
d) `repaint()`

7.10. Exerciţii propuse spre implementare

Exerciţiul 7.10.1. Presupunem că suntem în campanie electorală şi dorim să vizualizăm situaţia statistică a unui sondaj electoral. Presupunem că avem următorul caz:

- partidul P1 – 42%
- partidul P2 – 12%
- partidul P3 – 8%
- partidul P4 – 6%
- partidul P5 – 5%
- celelalte partide – 27%

Realizaţi un applet care să pună în evidenţă aceste proporţii. De exemplu, un cerc care să conţină diferite culori pentru anumite sectoare de cerc, reprezentate proporţional cu procentele furnizate în urma sondajului electoral.

Exerciţiul 7.10.2. Scrieţi un program Java care să convertească mile în kilometri (şi invers).

Exerciţiul 7.10.3. Construiţi un applet care să permită micşorarea/mărirea lungimii catetelor unui triunghi dreptunghic folosind două bare de defilare. De asemenea, afişaţi şi tangenta unui unghi ascuţit (*tan(alfa) = cateta1/cateta2*).

Exerciţiul 7.10.4. Scrieţi un applet care să afişeze 10 butoane corespunzătoare cifrelor zecimale şi un buton corespunzător trimiterii datelor şi validării ghicirii combinaţiei de cifre secrete setate prin program. (Este vorba, de fapt, de o parolă la începutul execuţiei programului.)

Exerciţiul 7.10.5. Implementaţi jocul X şi 0. Se va da posibilitatea de joc între utilizator şi calculator.

Exerciţiul 7.10.6. Utilizând componenta `TextArea` şi un buton, scrieţi un program Java care calculează numărul de caractere şi de linii din text.

Exerciţiul 7.10.7. Scrieţi un program Java care creează o arie text şi butoane pentru operaţiile de *cut-copy-paste* (decupare, copiere şi lipire).

Exerciţiul 7.10.8. Construiţi un program Java care are două butoane şi un `TextField` pentru citirea unui fişier grafic sau audio. Butoanele sunt corespunzătoare apelurilor `getImage()` şi `getAudioClip()`. Programul trebuie să citească numele fişierului şi să-l afişeze (să-l interpreteze).

Exerciţiul 7.10.9. Definiţi un `Frame` care generează un alt `Frame` mai mic, care la rândul său generează alt `Frame` şi mai mic, şi tot aşa (maximum 6 iteraţii). Fiecare `Frame` are un buton a cărui apăsare implică generarea următorului `Frame`.

Exerciţiul 7.10.10. Creaţi un applet care permite inserarea unei adrese a unei pagini Web. La acţionarea unui buton se va accesa pagina indicată. Se va da posibilitatea utilizatorului, printr-un buton radio, să aleagă dacă pagina va fi afişată în aceeaşi fereastră a navigatorului Web sau într-una nouă.

Exerciţiul 7.10.11. Modificaţi exemplul 7.3.3 astfel încât appletul să fie şi aplicaţie de sine stătătoare.

Exerciţiul 7.10.12. Modificaţi exemplul 7.4.5 înlocuind afişarea textuală a aruncării zarurilor cu o reprezentare grafică adecvată. Practic se vor desena feţele zarurilor obţinute în urma aruncării.

7.11. Proiecte propuse spre implementare

Proiectul 7.11.1. Scrieţi un program Java (care să fie simultan applet şi aplicaţie de sine stătătoare) care să simuleze un calculator de buzunar.

Proiectul 7.11.2. Să se construiască o aplicaţie care permite crearea de rebusuri. Fiecare rebus va fi salvat într-un fişier separat şi va putea fi utilizat de proiectul 7.11.3.

Proiectul 7.11.3. Să se realizeze o aplicaţie care să permită rezolvarea de rebusuri. Prin intermediul unei liste de opţiuni se va alege rebusul dorit. Fiecare rebus este definit într-un fişier creat de proiectul precedent.

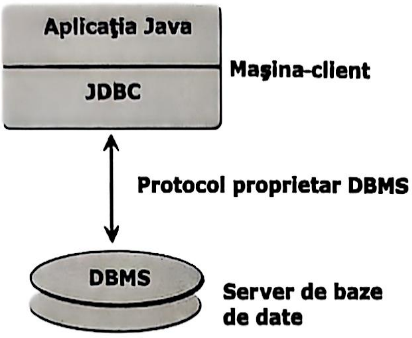
8. Accesul la baze de date folosind JDBC

8.1. Cuvinte-cheie

- baze de date
- DBMS (*DataBase Management System*)
- drivere JDBC
- interogări SQL
- mulţimi-rezultat – *ResultSet*
- tranzacţii

8.2. Introducere

JDBC (engl. *Java DataBase Connectivity*) reprezintă API-ul (engl. *Application Programming Interface*) dezvoltat de Sun Microsystems, în colaborare cu diverşi parteneri, pentru a oferi aplicaţiilor Java acces la bazele de date gestionate de diverse DBMS-uri (*DataBase Management System*, rom. Sisteme de Gestiune a Bazelor de Date - SGBD). Într-o arhitectură client-server, bazele de date se pot afla pe aceeaşi maşină sau pe o altă maşină cu care clientul este conectat dintr-un intranet sau chiar internet. Cele mai importante facilităţi ale JDBC-ului sunt posibilitatea lucrului cu instrucţiuni SQL (engl. *Structured Query Language*) şi procesarea eventualelor rezultate obţinute în urma interogărilor într-o manieră independentă şi consistentă. Putem privi API-ul JDBC ca o interfaţă standard între aplicaţii şi diverse DBMS-uri, creatorii aplicaţiilor programând, conform acestui API, într-un mod uniform şi independent de DBMS. Urmează ca apelurile SQL să fie preluate, traduse şi trimise mai departe de diverse drivere scrise în parte pentru fiecare DBMS.



Aşa cum se poate observa în figura precedentă, prin introducerea unui nivel de abstractizare reprezentat de API-ul JDBC (această metodă de lucru este des întâlnită în limbajul Java), se evită situaţia neplăcută ca programatorii să gestioneze singuri apelurile SQL către un DBMS particular, în acest caz codul aplicaţiei trebuind modificat pentru a se putea accesa un alt DBMS. Prin introducerea JDBC, pentru a ne conecta la un alt DBMS, este de ajuns să schimbăm driverul, operaţie care se poate realiza dinamic, chiar în timpul rulării aplicaţiei, nemaifiind necesară recompilarea acesteia. Această facilitate, împreună cu portabilitatea limbajului Java, permite interogarea oricărei baze de date de sub orice platformă fără recompilări de cod.

Maniera în care driverele trebuie construite este standardizat prin intermediul specificaţiilor JDBC care precizează interfeţele standard ce trebuie implementate de dezvoltatorii driverelor. De-a lungul timpului, s-a înregistrat o evoluţie a acestor specificaţii, în sensul creşterii funcţionalităţilor pe care driverele le pun la dispoziţie utilizatorilor, fără a afecta compatibilitatea cu versiunile anterioare ale specificaţiilor. În general, specificaţiile JDBC descriu o serie de interfeţe pe care cei ce dezvoltă driverele trebuie să le implementeze. Unele DBMS-uri nu permit folosirea procedurilor stocate sau a altor mecanisme datorită dezvoltării într-un mod nestandard a pieţei bazelor de date. De aceea, specificaţiile JDBC apărute de-a lungul timpului îi obligă pe cei care creează drivere la implementarea unei liste restrânse de interfeţe (eventual numai a anumitor metode ale lor), implementarea altora rămânând opțională, nerestrângând astfel posibilitățile reale ale DBMS-urilor deja existente.

Începând cu specificația `JDBC 2.0`, `API`-ul este împărțit în două părți: `JDBC 2.1 API`, care reprezintă nucleul API-ului JDBC, cuprins în pachetul `java.sql`, și `JDBC 2.0 Optional Package`, care oferă capabilități pe partea de server, cum ar fi tranzacții distribuite și conexiuni multiple, fiind conținut în pachetul `javax.sql`. Ultima specificație JDBC existentă în formă finală (engl. *final release*) are versiunea 3.0 și o puteți găsi la adresa `java.sun.com/products/jdbc`. Nu aduce diferențe majore față de precedenta versiune, 2.0, ci o prezintă pe aceasta într-o formă compactă și completând unele aspecte legate de aplicațiile distribuite. De aceea, conținutul acestui capitol corespunde nucleului versiunii 2.0. Această specificație aduce multe îmbunătățiri față de mai vechea distribuție, 1.0, cum ar fi folosirea cursoarelor, suport pentru stocarea obiectelor, acces programatic la bazele de date.

Pentru început putem spune că JDBC furnizează acces orientat pe obiecte la bazele de date, prin definirea de clase și interfețe care înfășoară diverse concepte abstracte, precum cele prezentate în tabelul următor:

Concepte înfășurate	Clasa/clasele sau interfețele corespunzătoare
Conexiuni la baze de date	`Connection`
Interogări SQL	`Statement, PreparedStatement,` `CallableStatement`
Mulțimi-rezultat	`ResultSet`
Obiecte mari binare sau caracter	`Blob (engl. Binary Large OBjects)` `Clob (engl. Character Large Objects)`
Drivere	`Driver`
Gestionari de drivere	`DriverManager`

De asemenea, standardul JDBC definește o serie de interfețe care trebuie implementate de creatorii driverelor cu scopul de a oferi dezvoltatorilor informații despre baza de date interogată, DBMS-ul folosit etc. Vom numi aceste informații *metadate* în sensul de „date despre date".

Interfețele puse la dispoziție pentru obținerea metadatelor le vom folosi în exemplele acestui capitol și le prezentăm în tabelul care urmează:

Interfață	Descriere
`DatabaseMetaData`	Interfață folosită de către cei care produc drivere pentru a informa utilizatorii despre capabilitățile oferite de DBMS împreună cu driverul JDBC folosit.
`ParameterMetaData`	Interfață folosită pentru a obține informații despre tipurile și proprietățile parametrilor dintr-un obiect `PreparedStatement`.
`ResultSetMetaData`	Interfață folosită pentru a obține informații despre tipurile și proprietățile coloanelor dintr-un `ResultSet`.

Ceea ce trebuie remarcat este faptul că aplicaţiile care folosesc bazele de date trebuie să includă pachetul `java.sql`, şi nu pachetul care conţine implementarea driverului particular folosit. Totuşi, calea spre pachetul conţinând driverul trebuie să fie prezentă în `CLASSPATH`.

8.3. Clasificarea driverelor JDBC

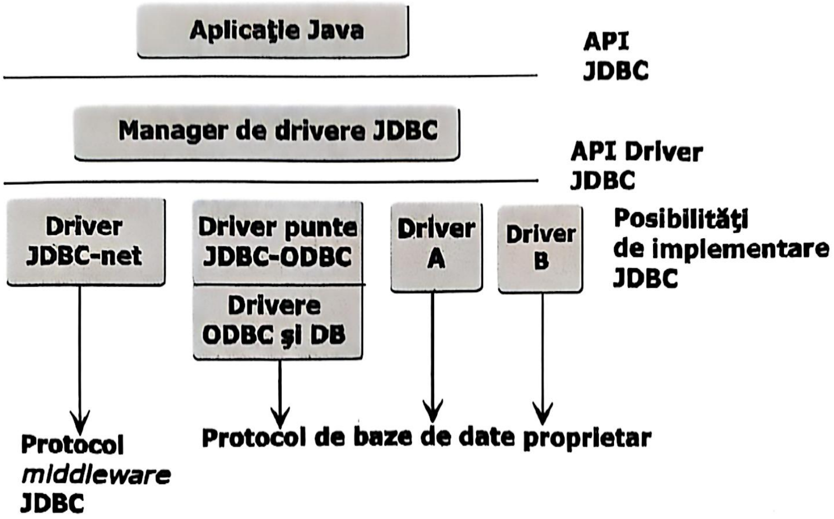
Sub aspectul funcţionalităţii, există patru tipuri de drivere JDBC:

1. Puntea JDBC-ODBC – acţionează ca o legătură dintre JDBC şi un alt mecanism de conectivitate a bazelor de date numit ODBC (engl. *Object DataBase Connectivity*). Acest driver vine odată cu ultimele distribuţii ale platformelor Java şi este reprezentat în Java 2 SDK de clasa `sun.jdbc.odbc.JdbcOdbcDriver`. Este folositor în aplicaţii de accesare a datelor unde nu există drivere JDBC pure. Puntea traduce metodele JDBC în apeluri de funcţii ODBC. Puntea JDBC-ODBC necesită ca bibliotecile ODBC native şi driverele ODBC să fie instalate şi configurate pentru fiecare client ce foloseşte un driver de tip 1. Această cerinţă reprezintă o limitare serioasă pentru multe aplicaţii, funcţionând doar pentru sistemele de operare *Microsoft Windows*, *Sun Solaris* şi *Linux*. Mai multe informaţii puteţi găsi în secţiunea acestui capitol dedicată utilizării punţii JDBC-ODBC.
2. Java-API nativ – aceste drivere folosesc interfaţa nativă Java (engl. JNI – *Java Native Interface*) pentru a face apeluri direct la API-ul unei baze de date locale. De obicei, driverele de tip 2 sunt mai rapide decât cele de tip 1. Ca şi driverele de tip 1, driverele de tip 2 necesită instalarea şi configurarea bibliotecilor native de accesare a datelor pe maşina-client. Sunt foarte convenabile când există biblioteci de accesare a datelor scrise în limbajul C, însă acestea nu sunt portabile pe toate platformele. Driverele de tip 2 sunt dedicate unui singur DBMS.
3. Java-protocol de reţea – acest tip include drivere Java pure care folosesc un protocol de reţea (de obicei, TCP/IP) pentru a comunica cu aplicaţia JDBC *middleware*. Apoi, aplicaţia JDBC *middleware* traduce cererile JDBC folosind protocolul de reţea în apeluri de funcţii specifice bazelor de date. Tipul 3 de drivere reprezintă soluţia cea mai flexibilă, deoarece nu necesită biblioteci de baze de date native pe client şi se poate conecta la mai multe baze de date diferite.
4. Java-protocol bază de date – acest tip conţine drivere Java pure care implementează un protocol de bază de date (cum ar fi *Oracle SQL Net*) pentru a comunica direct cu baza de date. De aceea, aceste drivere sunt cele mai rapide. Ca şi driverele de tip 3, ele nu necesită biblioteci de bază de date native şi pot fi folosite distribuit fără instalarea clientului. Driverele de tip 4 sunt specifice unui singur DBMS. Mai multe informaţii puteţi găsi în secţiunea 8.7.

În general, driverele de tip 1 şi 2 sunt gratuite, însă necesită instalarea la nivelul client a bibliotecile ODBC native, fiind folosite doar acolo unde nu sunt încă disponibile soluţii dezvoltate integral în Java. Driverele de tip 3 sunt, în general, comerciale şi se pot folosi în aplicaţii flexibile pe mai multe platforme. Driverele de tip 4 sunt mai rapide, unele sunt chiar gratis, însă se pot folosi doar pentru DBMS-uri particulare. Driverele de tipul 3 şi 4 oferă toate avantajele tehnologiei Java, printre care posibilitatea download-ului acestora în timp ce aplicaţia este folosită.

Puteţi afla lista driverelor JDBC din cele patru tipuri pentru diverse DBMS-uri şi conformitatea lor cu diversele specificaţii existente, pe situl firmei SUN la adresa `industry.java.sun.com/products/jdbc/drivers`. Se observă faptul că distribuţiile DBMS neproprietare oferă, în general, suport mai scăzut pentru Java decât cele proprietare, care au în general drivere conforme chiar cu specificaţia 3.0.

Pentru gestiunea driverelor folosite într-o aplicaţie, JDBC API foloseşte un manager de drivere reprezentat de o instanţă a clasei `DriverManager`. Managerul de drivere este capabil să suporte drivere concurente multiple conectate la baze de date eterogene multiple. În figura care urmează sunt prezentate aspecte ale conectării la baze de date:



8.4. Breviar SQL

SQL (engl. *Structured Query Language*) a devenit standard oficial pentru accesul la baze de date relaţionale în 1980. Motivul acceptării SQL-ului ca limbaj de interogare relaţional a fost, în principal, apariţia arhitecturilor client/server. În ceea ce priveşte interogările SQL, orice driver JDBC trebuie să suporte cel puţin secţiunea *Entry Level* a standardului adoptat în 1992 de ANSI, care poartă numele *SQL92* (*Entry Level* reprezintă o serie de funcţionalităţi SQL specificate de acest standard). Trebuie remarcat faptul că driverele conforme cu JDBC 2.0 oferă suport şi pentru tipuri *SQL3*.

Instrucţiunile SQL se împart în două categorii:

1. Instrucţiuni ale limbajului de definire a datelor (engl. *DDL – Data Definition Language*). Sunt folosite pentru crearea şi modificarea obiectelor bazei de date;
2. Instrucţiuni ale limbajului de manipulare a datelor (engl. *DML – Data Manipulation Language*). Sunt folosite pentru operaţii asupra datelor unei baze de date. Acestea pot fi împărţite în două categorii:

a) Instrucţiunea `SELECT` – care întoarce o mulţime de date rezultat ce satisfac un criteriu de interogare;
b) Alte instrucţiuni – care nu întorc o mulţime-rezultat.

Pentru crearea unei tabele (structura unei baze de date), vom folosi `DDL`, care defineşte sintaxa pentru comenzile: `CREATE TABLE` şi `ALTER TABLE`.

Pentru adăugarea, modificarea, ştergerea, căutarea datelor într-o bază de date se foloşeste limbajul DML (instrucţiunile `INSERT`, `UPDATE`, `DELETE`, `SELECT`).

Prezentăm în continuare o descriere amănunţită a principalelor instrucţiuni SQL, care vor fi folosite pe parcursul acestui capitol. Pentru un tutorial SQL, puteţi consulta `www.w3schools.com/sql`.

8.4.1. Instrucţiunea **CREATE TABLE**

Sintaxa generală a unei comenzi `CREATE TABLE` este:

```
CREATE TABLE <numetabela> (
    <numecamp> <tipdatasql> <descrieretip>,
    <numecamp> <tipdatasql> <descrieretip>,
  . . .
    <numecamp> <tipdatasql> <descrieretip>
 );
```

unde `<tipdatasql>` poate fi: `char`, `varchar`, `boolean`, `smallint`, `integer`, `numeric`, `float`, `currency`, `double`, `date`, `time`, `datetime`, `raw`. `<descrieretip>` se referă la caracteristicile tipului de dată SQL şi poate fi numărul maxim de caractere, dacă este cheie primară etc.

Exemplu:

```
CREATE TABLE studenti (
  nr int not NULL auto_increment primary key,
  nume varchar(25) NULL,
  prenume varchar(25) NULL,
  nota int NULL
)
```

8.4.2. Instrucţiunea **INSERT**

Definirea unei instrucţiuni `INSERT` implică trei părţi:

1. definirea tabelei de destinaţie pentru inserarea datelor;
2. definirea câmpurilor (coloanelor) care vor primi valori;
3. definirea valorilor pentru aceste coloane.

Astfel, sintaxa generală este:

```
INSERT INTO <numetabela> (camp1, camp2, . . ., campn)
          VALUES (valcamp1, valcamp2, . . ., valcampn);
```

Acţiunea acestei instrucţiuni SQL va fi adăugarea unei înregistrări ale cărei câmpuri sunt iniţializate cu valorile precizate biunivoc.

Exemplu:

```
INSERT INTO studenti (nr, nume, prenume, nota)
VALUES (1, Popescu, Ion, 7);
INSERT INTO studenti (nume, prenume) VALUES (Ionescu, Ion);
-- studentul Ionescu Ion va primi nota mai tarziu !
```

Dacă se doreşte completarea tuturor câmpurilor unei înregistrări, atunci se poate omite partea a doua, adică lista câmpurilor. Astfel, putem rescrie instrucţiunea de mai sus:

Exemplu:

```
INSERT INTO studenti values (1, Popescu, Ion, 7);
```

8.4.3. Instrucţiunea UPDATE

Definirea unei instrucţiuni `UPDATE` presupune trei paşi:

1. definirea tabelei de destinaţie pentru modificarea datelor;
2. definirea câmpului şi a unei noi valori;
3. definirea filtrului tabelei.

Astfel, sintaxa generală este:

```
  UPDATE <numetabela> SET camp1 = valcamp1
  WHERE camp2 = valcamp2;
```

Acţiunea acestei instrucţiuni SQL va fi modificarea (înlocuirea) unei valori a unui câmp dintr-o înregistrare.

Exemplu:

```
UPDATE studenti SET nota = 8 WHERE nr = 2;
```

8.4.4. Instrucţiunea DELETE

Definirea unei instrucţiuni `DELETE` se realizează în doi paşi:

1. definirea tabelei de destinaţie pentru ştergerea datelor;
2. definirea filtrului tabelei.

Astfel, sintaxa generală este:

```
DELETE FROM <numetabela> WHERE <conditie>;
```

Acţiunea acestei instrucţiuni SQL va fi ştergerea unor înregistrări.

Exemplu:

```
DELETE FROM studenti WHERE nr = 2;
```

8.4.5. Instrucţiunea SELECT

Se utilizează pentru a obţine înregistrări ale bazei de date care îndeplinesc anumite criterii. Există patru părţi ale acestei instrucţiuni:

1. definirea a ceea ce dorim să obţinem;
2. definirea tabelei de unde se vor obţine informaţiile;
3. definirea condiţiilor de obţinere (tabele `join`, filtre pentru înregistrări);
4. definirea ordinii în care se doreşte vizualizarea datelor.

Rezultatul întors de instrucţiunea `SELECT` este o mulţime de înregistrări (engl. `ResultSet`). Aşadar, un `ResultSet` poate fi privit ca o tabelă de date cu număr fixat de linii şi coloane.

Sintaxa generală a unei instrucţiuni `SELECT` este:

```
SELECT  <listacampuri> [AS <alias>] FROM <numetabela> WHERE
<expresiebooleana>;
```

În vederea obţinerii unei selectări cu `JOIN` (submulţime a unui produs cartezian), partea `<listacampuri>` depinde de `<numebazadate>`.

Exemple:

```
-- obtinerea listei studentilor promovati
SELECT nume, prenume AS porecla FROM studenti WHERE nota > 5;
-- daca se doreste selectarea tuturor inregistrarilor,
-- atunci se foloseste '*'
SELECT * FROM studenti;
-- presupunem ca avem trei baze de date specifice bibliotecii facultatii
-- de informatica. o baza se numeste autori şi are structura (nr, nume,
-- prenume), alta baza se numeste carti, cu structura (isbn, titlu,
-- bucati) si ultima baza de date se numeste autoricarti cu structura
-- (nr, isbn). Dorim sa selectam o lista de autori si cartile lor
-- corespunzatoare.
SELECT a.nume, a.prenume, b.titlu FROM autori a, carti b,
autoricarti ab WHERE a.nr = ab.nr AND b.isbn = ab.isbn;
```

8.5. Asocieri de tipuri între SQL şi Java

Tipurile de date din SQL sunt diferite de tipurile de date suportate de limbajul Java. De asemenea, există deosebiri semnificative între tipurile suportate de diferite DBMS-uri existente pe piaţă. Sau chiar dacă practic acestea suportă tipuri cu aceeaşi semantică, tipurile poartă alte nume. Spre exemplu, tipul `BYTE` din Access 7.0 corespunde tipului `TINYINT` din Sybase 11.9 şi ambele se referă la un întreg din intervalul [0, 255], eventual cu semn. Pentru a evita aceste incompatibilităţi, JDBC defineşte o serie de tipuri generice SQL prin intermediul clasei `java.sql.Types`, permiţând programatorului să nu se mai îngrijească de numele tipurilor de date folosite de DBMS-ul la care se conectează. Trecerea între tipurile generice JDBC şi tipurile efective, caracteristice bazelor de date, cade în sarcina driverului. Programatorul interacţionează în aplicaţia lui numai cu tipurile JDBC, fiind definită o asociere standard între tipurile JDBC şi tipurile Java. Acest aspect determină o interfaţă simplă pentru scrierea şi citirea de valori ca simple tipuri Java. Tabelul următor prezintă asocierea JDBC-Java:

Tip JDBC	Tip Java
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	Boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	double
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
CLOB	Clob
BLOB	Blob
ARRAY	Array
DISTINCT	tip corespunzator
STRUCT	Struct
REF	Ref
JAVA_OBJECT	clasa JAVA corespunzatoare

Un programator trebuie să ţină cont de acest tabel atunci când scrie aplicaţia care va interacţiona cu baza de date. Spre exemplu, dacă în unul dintre tabelele interogate avem un câmp de tip `REAL`, atunci va trebui să folosească o variabilă de tip `float` pentru a stoca informaţiile din celulele acelei coloane.

Locul în care programatorii interacţionează totuşi cu tipuri de date folosite de DBMS îl reprezintă instrucţiunile SQL de tip `CREATE TABLE`, pe care aceştia le folosesc pentru a crea tabele. În acest caz, este indicată citirea cu atenţie a documentaţiei DBMS-ului. O modalitate de a scrie totuşi comenzi `CREATE TABLE` care să funcţioneze pentru o varietate de baze de date este folosirea unui număr redus de tipuri de date, cum ar fi `INTEGER`, `NUMERIC` sau `VARCHAR`, unanim acceptate de toate DBMS-urile. Un alt mod ar fi folosirea metodei `java.sql.DatabaseMetaData.getTypeInfo()` pentru a descoperi tipurile de date acceptate de DBMS.

În continuare, prezentăm mecanismele pentru transferarea datelor între aplicaţii care folosesc tipuri Java şi bazele de date interogate care folosesc tipuri SQL. Aceste metode le veţi regăsi folosite în secţiunile următoare.

Metode-interacţiune	Forma generică	Observaţii
Metode aplicate `ResultSet`-ului obţinut în urma executării interogărilor SQL.	Metode de forma `getXXX`, unde `XXX` reprezintă tipul de date preferat. În cazul JDBC 2.0 apar şi metode de forma `updateXXX()`.	Mai general, metoda `getString()` poate fi folosită pentru a returna orice tip de date cu precizarea că cele care aveau tipul numeric trebuie reconvertite. De asemenea, în acelaşi scop se pot folosi apeluri `getObject()` cu operatorii `cast` corespunzători.
Metode aplicate instanţelor `PreparedStatement` pentru a da valori instrucţiunilor parametrizate.	Metode de forma `setXXX`, unde `XXX` reprezintă tipul de date preferat.	`XXX` reprezintă un tip Java oarecare.
Metode aplicate instanţelor `CallableStatement` pentru a obţine parametrii de ieşire.	Metode de forma `getXXX`, unde `XXX` reprezintă tipul de date preferat.	Se poate folosi, de asemenea, `getObject()`, împreună cu operatorii `cast` corespunzători.

JDBC 2.0 aduce noi tipuri de date cunoscute ca tipuri SQL3. Corespunzător acestor tipuri, JDBC oferă interfeţe pentru asocierea lor. Prezentăm în continuare tipurile SQL, descrierea lor şi interfeţele JDBC corespunzătoare:

Tip SQL3	Interfaţă JDBC	Descriere
`BLOB`	`Blob`	Oferă posibilitatea stocării unei cantităţi mari de date reprezentate sub formă de şir de biţi (binară).
`CLOB`	`Clob`	Oferă posibilitatea stocării unei cantităţi foarte mari de date în format caracter.
`ARRAY`	`Array`	Se pot stoca tablouri.
Tip structurat SQL	`Struct`	Permite folosirea tipurilor de date structurate SQL definite de utilizator.
`REF`	`Ref`	Permite accesarea prin referinţă a tipurilor de date structurate SQL, definite de utilizator.

Pentru a citi, stoca şi actualiza date de tip SQL3, se vor folosi metode de forma `getXXX`, `setXXX` şi `updateXXX`, unde `XXX` reprezintă tipul corespunzător coloanei a doua din tabelul de mai sus.

SQL permite şi definirea de tipuri de date utilizator, cunoscute şi sub numele de UDT (engl. *User Defined Types*) prin intermediul instrucţiunilor SQL `CREATE TYPE`. Acestea se împart în tipuri structurate şi tipuri distincte. Spre exemplu, prezentăm modul de creare a unui tip structurat de date:

```
        CREATE TYPE PUNCT_PLAN
(
                X FLOAT,
                Y FLOAT
)
```

Un tip distinct poate fi creat având la bază un tip deja existent, aşa cum se poate vedea în exemplul următor:

```
CREATE TYPE NUMERE AS NUMERIC(10, 2)
```

Prin definiţie, un tip distinct SQL este asociat aceluiaşi tip ca şi tipul de bază. În exemplul prezentat, deoarece tipul `NUMERIC` este asociat cu `java.math.BigDecimal`, aceeaşi asociere va căpăta şi tipul `NUMERE`. Deci vom volosi `ResultSet.getBigDecimal()` pentru a obţine o valoare de acest tip.

8.6. Accesarea unei baze de date folosind JDBC

Procesul obţinerii de informaţii dintr-o bază de date folosind JDBC implică în principiu cinci paşi:

1. înregistrarea driverului JDBC folosind gestionarul de drivere `DriverManager`;
2. stabilirea unei conexiuni cu baza de date;
3. execuţia unei instrucţiuni SQL;
4. procesarea rezultatelor;
5. închiderea conexiunii cu baza de date.

8.6.1. Înregistrarea driverului JDBC folosind clasa `DriverManager`

Rolul managerului de drivere este acela de a ţine o referinţă la fiecare dintre obiectele driver disponibile în aplicaţia curentă. Un driver JDBC este înregistrat automat de managerul de drivere atunci când clasa driver este încărcată dinamic. Pentru încărcarea dinamică a unui driver JDBC, folosim metodele `Class.forName()`. După cum se observă, nu este nevoie de crearea unei instanţe a clasei driver odată ce aceasta a fost încărcată. Dacă se apelează metoda `newInstance()`, se va realiza un duplicat nefolositor al clasei driver. În exemplul următor vom încărca driverul punte JDBC-ODBC din pachetul `sun.jdbc.odbc` şi, mai apoi, driverul MySQL Connector/J, care permite conectarea la serverul de baze de date MySQL. Trebuie remarcat faptul că pentru Connector/J este necesar să includem în `CLASSPATH` calea spre pachetul care conţine driverul.

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"));
Class.forName("com.mysql.jdbc.Driver");
```

`Class.forName()` este o metodă statică. Aceasta permite maşinii virtuale Java să aloce dinamic, să încarce şi să facă o legătură la clasa specificată ca argument printr-un şir de caractere. În cazul în care clasa nu este găsită, se aruncă o excepţie `ClassNotFoundException`. A se vedea `Java Reflection API`.

Driverele pot fi înregistrate şi folosind metoda `DriverManager.registerDriver()`.

8.6.2. Stabilirea unei conexiuni cu baza de date

Odată ce s-a încărcat un driver, putem să-l folosim pentru stabilirea unei conexiuni cu baza de date. O conexiune JDBC este identificată printr-un URL JDBC specific. Sintaxa standard pentru URL-ul unei baze de date este:

```
jdbc:<subprotocol>:<nume>
```

Prima parte precizează că pentru stabilirea conexiunii se foloseşte JDBC. Partea de mijloc `<subprotocol>` este un nume de driver valid sau al altei soluţii de conexiune a bazelor de date. Ultima parte, `<nume>`, este un nume logic sau alias care corespunde bazei de date fizice. Dacă baza de date va fi accesată prin internet, secţiunea `<nume>` va respecta următoarea convenţie de nume:

```
//numegazda:port/subsubnume
```

În acest sens, un exemplu de adresă corectă este:

```
jdbc:mysql://localhost:386/arhiva
```

Prezentăm în continuare sintaxa particulară pentru câteva drivere JDBC:

ODBC - `jdbc:odbc:<sursa_date>[;<nume_atribut>=<valoare_atribut>]*`
MySQL - `jdbc:mysql://server[:port]/numeBazaDate`
Oracle - `jdbc:oracle:thin:@server:port:numeInstanta`

Pentru stabilirea unei conexiuni la o bază de date, se foloseşte metoda statică `getConnection()` din clasa `DriverManager`.

```
Connection con = DriverManager.getConnection(
   "jdbc:mysql://localhost/arhiva");
```

Pentru baze de date care necesită autentificare, se utilizează o formă a acestei metode cu trei argumente.

```
Connection con = DriverManager.getConnection(
   "jdbc:mysql://localhost/arhiva", nume_utilizator, parola);
```

Pentru a afla informaţii despre DBMS, va trebui să apelăm `Connection.getMetaData()` pentru a obţine o instanţă `DatabaseMetaData`, căreia îi putem aplica diverse metode de a afla informaţii despre tabelele bazei de date, sintaxa SQL suportată, dacă suportă sau nu proceduri stocate etc.

Folosind JNDI (*Java Name and Directory Interface*) putem să ne conectăm la o bază de date considerând-o sursă de date având un nume logic, în loc să codăm direct în aplicaţie numele ei şi driverul pe care îl vom folosi. Acest mod de conectare nu este obiectul acestui capitol, fiind o caracteristică a sistemelor distribuite.

8.6.3. Execuţia unei instrucţiuni SQL

După ce s-a stabilit conexiunea, se pot trimite instrucţiuni SQL către baza de date. API-ul JDBC nu verifică dacă instrucţiunea este corectă şi nici apartenenţa ei la un anumit standard SQL, permiţându-se astfel trimiterea chiar de instrucţiuni nonSQL. Programatorul este cel care ştie dacă DBMS-ul interogat suportă interogările pe care le trimite şi, dacă nu, el este cel care va trata excepţiile primite drept răspuns. Dacă instrucţiunea este SQL, atunci aceasta poate face anumite operaţii asupra bazei de date, cum ar fi căutare, inserare, actualizare sau ştergere.

API-ul JDBC specifică trei interfeţe (pe care dezvoltatorul driverului trebuie să le implementeze) pentru trimiterea de interogări către bazele de date, fiecăreia corespunzându-i o metodă specială în clasa `Connection`, de creare a instanţelor corespunzătoare. Acestea sunt prezentate în tabelul care urmează:

Clasa	Metoda de creare	Explicaţii
`Statement`	`Connection.create Statement()`	Este folosită pentru trimiterea de instrucţiuni SQL simple, fără parametri.
`Prepared Statement`	`Connection.prepare Statement()`	Permite folosirea instrucţiunilor SQL precompilate şi a parametrilor de intrare în interogări. Această metodă de a face interogări este utilă în cazul în care se fac interogări care diferă doar printr-un număr de parametri.
`Callable Statement`	`Connection.prepare Call()`	Permite folosire procedurilor stocate pe serverul DBMS.

Pentru execuţia unei instrucţiuni SQL neparametrizate, se foloseşte metoda `createStatement()`, aplicată unui obiect `Connection`. Această metodă întoarce un obiect din clasa Statement.

```
Statement instructiune = con.createStatement();
```

Putem aplica apoi una dintre metodele `executeQuery()`, `executeUpdate()` sau `execute()` obiectului de tip `Statement` pentru a trimite DBMS-ului instrucţiunile SQL. Metoda `executeQuery()` este folosită în cazul interogărilor care returnează mulţimi-rezultat (instanţe ale clasei `ResultSet`), aşa cum este cazul instrucţiunilor `SELECT`. Pentru operaţiile de actualizare sau ştergere, cum ar fi `INSERT`, `UPDATE` sau `DELETE`, se foloseşte metoda `executeUpdate()` aplicată obiectului de tip `Statement`, rezultând un întreg care reprezintă numărul înregistrării afectate. Aceeaşi metodă este folosită pentru interogările SQL DDL, cum ar fi `CREATE TABLE`, `DROP TABLE` şi `ALTER TABLE`, în acest caz returnând întotdeauna zero. Metoda `execute()` este utilizată în cazul în care se obţine mai mult de o mulţime-rezultat sau un număr de linii.

```
ResultSet rs = instructiune.executeQuery(
"select * from arhive");
String sql = "insert into arhive values (
'Popescu', 'Ion', 'Iasi')";
int raspuns = instructiune.executeUpdate(sql);
```

JDBC 2.0 permite trimiterea spre execuţie a mai multor instrucţiuni SQL grupate (engl. *batch*), această facilitate mărind uneori performanţele. Prezentăm în continuare un astfel de caz:

```
Statement inter = con.createStatement();
con.setAutoCommit(false);

inter.addBatch("INSERT INTO arhive VALUES (1, 'Popescu', 'Ioan')");
inter.addBatch("INSERT INTO cantitati VALUES (260, 'file')");
inter.addBatch("INSERT INTO localitati VALUES ('iasi', 'oras')");
int [] actualizari = inter.executeBatch();
```

Se observă dezactivarea modului *autocommit*. Acest aspect determină ca modificările rezultate în tabele după execuţia metodei `executeBatch()` să nu fie automat salvate permanent (engl. *commit*) sau anulate (engl. *rollback*), aceste operaţiuni rămânând la alegerea clientului. Se poate astfel ca în cazul în care una dintre interogări eşuează, clientul să anuleze efectele tuturor. Pentru alte informaţii, puteţi consulta secţiunea dedicată tranzacţiilor. Metoda `executeBatch()` returnează un tablou în care elementele corespund interogărilor SQL efectuate şi reprezintă numărul de linii afectate de instrucţiunile SQL corespunzătoare.

Pentru a executa independent instrucţiunile SQL, se poate păstra modul *autocommit* folosind captarea excepţiilor `BatchUpdateException` aruncate în caz de eroare, aşa cum se poate vedea în exemplul următor:

```
try {
   inter.addBatch("INSERT INTO arhive VALUES (1, 'Popescu', 'Ioan')");
   inter.addBatch("INSERT INTO cantitati VALUES (260, 'file')");
   inter.addBatch("INSERT INTO localitati VALUES ('iasi', 'oras')");
   int [] actualizari = stmt.executeBatch();
} catch(BatchUpdateException b) {
     System.err.println("Actualizari realizate: ");
     int [] eroriActualizari = b.getUpdateCounts();
     for (int  i = 0; i < eroriActualizari.length; i++) {
          System.err.print(eroriActualizari[i] + "  ");
     }
     System.err.println("");
}
```

Ştergerea comenzilor dintr-un grup se realizează cu metoda `clearBatch()`.

Un driver JDBC poate să nu ofere suport pentru execuţia grupată a unor instrucţiuni SQL. Pentru a afla dacă e permis acest lucru, se va folosi metoda `supportsBatchUpdates()` din clasa `DatabaseMetaData`.

Dacă se doreşte realizarea de apeluri SQL având date variabile drept intrare, se va folosi clasa `PreparedStatement` care moşteneşte clasa `Statement`. Prezentăm în continuare modul de construcţie a unei astfel de instrucţiuni, unde `con` reprezintă o instanţă `Connection`:

```
PreparedStatement  instructiune = con.prepareStatement(
"update arhive set nume = ? where prenume like ?");
```

După cum se observă, prin construcţie, obiectul de tip `PreparedStatement` primeşte ca intrare o instrucţiune SQL (spre deosebire de cazul `Statement`). În momentul construcţiei, instrucţiunea SQL primită ca parametru este trimisă direct spre DBMS, unde este precompilată. Mai rămâne să dăm valori variabilelor folosind metode `setXX()` corespunzătoare tipurilor de date şi să executăm efectiv interogarea, după cum se poate vedea în continuare:

```
instructiune.setString(1,"Popescu");
instructiune.setString(2,"Ion");
instructiune.executeUpdate();
```

Această manieră de interogare este mai rapidă decât folosirea clasei `Statement`, în cazul în care dorim execuţia repetată a unei instrucţiuni SQL. Instrucţiunea SQL este compilată o singură dată în momentul creării instanţei clasei `PreparedStatement` şi folosită apoi repetat, eventual cu valori diferite ale parametrilor. Se poate folosi şi în cazul în care nu avem parametri, după cum se observă în continuare:

```
PreparedStatement  instructiune = con.prepareStatement(
"select * from arhive");
ResultSet rs = instructiune.executeQuery();
```

Nu vom prezenta în acest capitol procedurile stocate datorită nivelului relativ ridicat de complexitate.

Metodele fără parametri de intrare puse la dispoziţie de clasa `Connection` pentru cele trei interfeţe prezentate în tabelul precedent produc mulţimi-rezultat fără cursor deplasabil şi nesenzitive la modificări. O mulţime-rezultat are cursor pentru poziţionare deplasabil, proprietate care poartă numele de *scrolling*, dacă deţine un pointer interior care indică înregistrarea accesată la momentul curent, pointer ce se poate deplasa programatic. De asemenea, după cum spuneam, mulţimea-rezultat nu este senzitivă (engl. *sensitive*) la modificările care pot surveni între timp în tabela interogată. Prin senzitivitate se înţelege actualizarea automată a mulţimii-rezultat pentru a ilustra dinamic modificările survenite între timp în tabelă.

Driverele care sunt conforme cu versiunea 2.0 a specificaţiei JDBC permit obţinerea de mulţimi-rezultat senzitive şi având cursor deplasabil. Acest lucru este posibil prin specificarea în constructorul instanţei `Statement` a uneia dintre constantele prezentate în tabelul care urmează:

Constantă	Explicaţii
TYPE_FORWARD_ONLY	Acest tip de `ResultSet` este cel din JDBC 1.0. Mulţimea-rezultat nu are cursor deplasabil decât dinspre început spre sfârşit. Modificările făcute în tabelă nu se reflectă în mulţimea-rezultat.
TYPE_SCROLL_INSENSITIVE	Mulţime-rezultat *scrollable*, deci cursorul poate fi mutat înainte şi înapoi sau pe orice poziţie diferită de poziţia curentă. De asemenea, eventualele modificări făcute între timp în tabelă nu sunt reflectare în `ResultSet`.
TYPE_SCROLL_SENSITIVE	Mulţime-rezultat *scrollable*. Senzitivitate la modificări.

De asemenea, driverele conforme cu specificaţia 2.0 a JDBC oferă şi posibilitatea actualizării programatice a tabelelor prin intermediul obiectelor `ResultSet`. Tipul de actualizare se specifică prin intermediul uneia dintre constantele prezentate în continuare:

Constantă	Explicaţii
CONCUR_READ_ONLY	`ResultSet`-ul nu poate fi modificat programatic. Oferă posibilitatea unui număr nelimitat de conectări concurente la baza de date, deoarece nu se fac modificări asupra tabelei care ar putea genera conflicte. Acest tip de `ResultSet` este specific driverelor conforme specificaţiei JDBC 1.0.
CONCUR_UPDATABLE	`ResultSet`-ul şi baza de date pot fi modificate programatic. Sunt posibile un număr limitat de conexiuni concurente dat de tipul de concurenţă ales (a se vedea secţiunea dedicată tranzacţiilor). Acest tip de `ResultSet` este specific driverelor conforme specificaţiei JDBC 2.0.

Prezentăm un exemplu de cod care va determina crearea unei mulţimi-rezultat având cursor deplasabil şi senzitivă la modificări (ordinea parametrilor metodei `createStatement()` este importantă).

```
Statement declar = con.createStatement(
   ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.
   CONCUR_UPDATABLE);
// la fel se poate crea o instanta a clasei PreparedStatement
ResultSet rez = declar.executeQuery(
   "select nume, prenume from arhive");
```

8.6.4. Procesarea rezultatelor

Pentru parcurgerea simplă a înregistrărilor unui obiect din clasa `ResultSet` folosind JDBC 1.0, putem folosi metoda `next()`, ca în următoarea secvenţă de cod.

```
while (rs.next()) // implicit, cursor pozitionat inainte de prima
```

```
// linie
{
   System.out.println(rs.getString("nume") + ", " +
     rs.getString("prenume") + ", " + rs.getString("oras"));
}
```

În cazul JDBC 2.0, odată obţinută, mulţimea-rezultat `rez` (a se vedea paragraful precedent), având cursor deplasabil, poate fi folosită pentru a poziţiona cursorul în interiorul `ResultSet`-ului. Iniţial, cursorul este poziţionat înaintea primei linii din mulţimea-rezultat. În JDBC 1.0, după cum aminteam, cursorul putea fi mutat doar înainte folosind metoda `next()`. Driverele care sunt conforme cu JDBC 2.0 mai permit poziţionarea absolută a cursorului pe orice linie, folosind metoda `absolute (int nr_linie)`, poziţionări relative ale cursorului relativ la linia curentă folosind `relative(int nr_salt)`, respectiv deplasări înapoi folosind metoda `previous()`. Dacă argumentul metodei `absolute()` este pozitiv, se va realiza poziţionarea pe linia corespunzătoare. Dacă argumentul este negativ, poziţionarea se va face prin deplasare de la sfârşit spre început cu un număr de linii egal cu valoarea absolută din argument. Astfel, `absolute(1)` va determina poziţionarea pe prima linie, iar `absolute(1)` va determina întotdeauna poziţionarea pe ultima linie. Pentru determinarea liniei curente, se poate folosi metoda `getRow()`. De asemenea, există metodele `first()`, `beforeFirst()` pentru poziţionarea pe prima, respectiv înainte de prima linie, `last()` şi `afterLast()` pentru poziţionarea pe ultima, respectiv după ultima linie. Pentru testarea poziţiei, avem metodele `isFirst()`, `isLast()`, `isBeforeFirst()`, `isBeforeLast()`. De remarcat că toate poziţionările se referă la `ResultSet`, şi nu la baza de date interogată.

Prezentăm aceste metode folosite pentru a citi în ordinea inversă liniile din `ResultSet`-ul obţinut în urma interogării precedente:

```
if (rez.isAfterLast() == false) {
      rez.afterLast();
}
while (rez.previous()) {
    String nume = rez.getString(2);
// nume va detine continutul celulei aflata la intersectia liniei
// curente din ResultSet cu prima coloana tot din ResultSet
    String prenume = rez.getString("prenume");
// prenume va detine continutul celulei aflata la intersectia
// liniei curente din ResultSet cu coloana prenume din ResultSet
    System.out.println(rez.getRow() + "nume: " + nume + "
    prenume: " + prenume);
}
absolute(1); // pozitioneaza cursorul pe ultima linie
```

Dacă nu cunoaştem tipul coloanelor `ResultSet`-ului, putem folosi un apel `ResultSet.getMetaData()` pentru a obţine o instanţă `ResultSetMetaData`, pentru care, mai apoi, se aplică metoda `getColumnType()`.

În cazul în care driverul folosit este conform cu JDBC 2.0, clasa `ResultSet` oferă suport pentru actualizări programatice. Prin actualizări programatice vom înţelege actualizări aplicate `ResultSet`-ului, care sunt automat efectuate şi asupra bazei de date. În acest fel, baza de date va putea fi modificată fără a efectua instrucţiuni SQL. Metodele pe care clasa `ResultSet` le pune la dispoziţie pentru astfel de actualizări sunt `updateXXX()`, unde `XXX` reprezintă un tip de date Java. Spre exemplu, `updateString (String nume_camp, String valoare)`, poate fi folosită pentru a actualiza celula aflată la intersecţia dintre linia curentă şi coloana `nume_camp`. Metodele `getXXX` cu care se inspectează `ResultSet`-ul au corespondente `setXXX`.

```
rez.last();
rez.updateInt("nr_curent", 100);
rez.updateString(2,"Popescu");
rez.updateRow();
```

După cum se observă, celula care se vrea actualizată aparţine înregistrării curente din `ResultSet`, iar coloana este specificată fie prin indexul său din `ResultSet`, fie prin nume. Prin apelarea metodelor de tip `updateXXX()`, se realizează actualizarea numai în `ResultSet`. Pentru a face actualizarea şi în tabla încapsulată, se va folosi mai apoi metoda `updateRow()`. Este foarte important ca apelul acestei metode să se facă atunci când cursorul este poziţionat pe înregistrarea care se vrea actualizată. Dacă cursorul este deplasat înainte de apel, driverul va anula actualizarea şi, drept consecinţă, numai mulţimea-rezultat va fi modificată, nu şi baza de date.

Pentru a evita ca modificările făcute în `ResultSet` să aibă efect şi în baza de date, avem la dispoziţe metoda `cancelRowUpdates()`, care, pentru a avea efect, trebuie apelată între metodele `updateXXX` şi `updateRow()`. Dacă este apelată după metoda `updateRow()`, nu mai are nici o valoare.

Pe lângă actualizări, API-ul JDBC 2.0 oferă şi posibilitatea ştergerii de linii folosind metoda `deleteRow()`.

```
rez.last();
rez.deleteRow();
```

Ştergerea se aplică atât `ResultSet`-ului, cât şi tabelei înfăşurate. De observat că în exemplul precedent se va şterge ultima linie din `ResultSet`, şi nu din tabela încapsulată. Poziţia pe care se află în tabelă linia din `ResultSet` ce se vrea ştearsă rămâne întotdeauna nedeterminată.

Ultima operaţie pe care o vom prezenta este inserarea unei linii într-o tabelă. Această operaţie se realizează prin intermediul unei linii speciale care poartă numele *insert row*. Aceasta este o linie ajutătoare asociată unei mulţimi-rezultat, dar neconţinută de aceasta din urmă, folosind la completarea cu date a unei linii care va fi mai apoi inserată efectiv în tabelă. Pentru a obţine accesul la *insert row*, clasa `ResultSet` pune la dispoziţie metoda `moveToInsertRow()`. Urmează apelul metodelor de tipul `updateXXX` pentru a completa linia *insert row* cu informaţii. După ce linia a fost completată cu date, se va insera efectiv în tabela curentă printr-un apel `insertRow()`. Mai urmează repoziţionarea pe o înregistrare dorită din mulţimea-rezultat. Spre exemplu, pentru a ne repoziţiona pe înregistrarea pe care eram înainte de a insera o linie, se va apela metoda `ResultSet.moveToCurrentRow()`.

```
rez.moveToInsertRow();
rez.updateInt(1, 357);
rez.updateString(2, "Popescu");
rez.updateString(3, "Ioan");
rez.insertRow();
rez.first();
```

Apelurile de forma `updateXXX` efectuate când suntem poziţionaţi pe *`insert row`* nu au nici un efect asupra mulţimii-rezultat şi nici asupra bazei de date din care a fost generat `ResultSet`-ul. Apelul `insertRow()` va returna o excepţie `SQLException` dacă numărul de coloane din `ResultSet` este mai mic decât numărul de coloane din tabela încapsulată. De asemenea, putem obţine informaţii despre datele din `insert row` folosind metode `getXXX`, dar dacă nu am actualizat înainte celula executând un apel de forma `updateXXX`, ceea ce obţinem drept rezultat este nedefinit.

Atunci când inserăm o înregistrare în tabelă, JDBC nu oferă nici o modalitate de a afla pe ce poziţie a fost introdusă acea înregistrare. Acest lucru este gestionat intern de DBMS.

Pe lângă tipurile de date corespunzătoare specificaţiei SQL-92, prin intermediul unei instanţe `ResultSet` se poate lucra şi cu date având tipuri SQL3. Prezentăm în continuare un exemplu de citire a unui tablou dintr-o tabelă:

```
ResultSet rs = stmt.executeQuery(
"select rezultate from tabela where  nr_curent = 3");
rs.next();
Array tablou = rs.getArray("rezultate");
```

Un exemplu de stocare în mod programatic a unui obiect `Clob`:

```
Clob observatii = rs.getClob("observatii");
PreparedStatement prep = con.prepareStatement(
    "update alta_tabela set comentarii = ? where nr_curent
    < 1000",
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_UPDATABLE);
prep.setClob(1, observatii);
```

Dintr-un obiect `Clob` putem să obţinem un flux de caractere:

```
java.io.Reader citire = observatii.getCharacterStream();
```

Trebuie remarcat că în cazul tipurilor `CLOB`, `BLOB` şi `ARRAY`, prin interogări se obţin referinţe spre obiectele de acest tip stocate în bazele de date, în loc ca acestea să fie aduse din DBMS în aplicaţie, soluţia fiind adoptată pentru optimizare. Cu toate acestea, folosirea acestor tipuri – din cauza cantităţii mari de date conţinute – determină scăderi în ceea ce priveşte performanţele. Accesul spre datele de tip `BLOB` şi `CLOB` stocate în baza de date se realizează prin fluxuri de citire/scriere. Prezentăm în continuare operaţia de citire folosind un flux:

```
String sir = "";
java.sql.Blob blob = rezultat.getBlob(i);
InputStream inn = blob.getBinaryStream();
InputStreamReader in = new InputStreamReader(inn);
int b;
while (in.read()> 1) {
  b = in.read();
  sir+ = b;
}
in.close();
inn.close();
```

În cazul tipurilor definite de utilizator (tipuri distincte şi structurate), prezentăm modalitatea de interogare şi de obţinere a rezultatului:

```
ResultSet rs = stmt.executeQuery(
     "SELECT PUNCT FROM FIGURI WHERE DIM > 3");
while (rs.next()) {
     Struct punct = (Struct)rs.getObject("PUNCT");
     // operatii cu obiectul punct
}
```

În ambele cazuri, şi pentru tipuri distincte, şi pentru cele structurate, returnarea se face prin valoare, şi nu prin referinţă.

8.6.5. Închiderea unei conexiuni la o bază de date

Atunci când este vorba despre resurse exterioare, aşa cum este cazul accesului la DBMS-uri via JDBC, „colectorul de gunoaie" nu ştie nimic despre starea acestor resurse, dacă este sau nu cazul să le elibereze. De aceea, este recomandat ca, după ce procesarea datelor s-a încheiat, programatorii să închidă explicit conexiunile către baza de date. Pentru aceasta se foloseşte metoda `close()` aplicată obiectului `Connection`. În plus, trebuie închise (înaintea conexiunii) şi obiectele `Statement` şi `ResultSet` folosind metodele lor `close()`, ca în exemplul următor:

```
try  {
   //. . .
   rs.close();
   instructiune.close();
 }
 catch (SQLException e) {
   System.err.println("Eroare la inchiderea interogarii: " +
   e.toString());
 }
 finally {
   try  {
```

```
            if (conexiuneBazaDate != null)   {
                conexiuneBazaDate.close();
            }
        }
        catch (SQLException ex) {
        System.err.println("Eroare la inchiderea conexiunii: " +
        ex.toString());
    }
}
```

8.7. Instalarea şi configurarea MySQL

8.7.1. Introducere

Unul dintre cele mai simple şi cunoscute servere de baze de date este MySQL, care şi-a dovedit în timp stabilitatea şi fiabilitatea, fiind deseori soluţia aleasă pentru gestiunea bazelor de date. Serverul MySQL este oferit gratuit pentru sistemele Unix, Windows şi MacOS, sub licenţa *GNU General Public License* (GPL), de către firma suedeză MySQL AB. Situl oficial MySQL este *http://www.mysql.com/*, care poate fi consultat pentru a descărca ultimele distribuţii MySQL, deja stabile sau încă în probă, împreună cu documentaţia aferentă. Tot de aici se poate descărca şi driverul JDBC oficial, care poartă numele *Connector/J*, de fapt, mai vechiul *mmmysql* preluat sub tutela AB şi îmbunătăţit. În ceea ce priveşte documentaţia, manualul de utilizare MySQL pare a fi referinţa cea mai importantă pentru dezvoltatori.

După instalare, bazele de date pe care serverul le va pune la dispoziţie clienţilor vor fi poziţionate în directorul `/data` al distribuţiei. În subdirectorul `/bin` al distribuţiei MySQL se va găsi executabilul `mysqld`, care permite startarea serverului de baze de date. În acelaşi director se află şi alte utilitare, cum ar fi *WinMySQLadmin* pentru sistemul de operare Windows, care permite administrarea serverului MySQL dintr-o interfaţă grafică prietenoasă. Rularea programului *WinMySQLadmin* se poate face urmând paşii: clic-stânga pe butonul *Start*, apoi *Programs* şi *Startup*. După selectare, dacă fereastra-utilizator se minimizează în bara de stare a ecranului, atunci se apăsa clic-dreapta şi se selectează *Start the server*, urmat de *Show me*. Pentru vizualizarea bazelor de date active se selectează panoul *Databases*, în interiorul căruia se apăsa clic-dreapta. Apoi se selectează *Create Database* din meniul contextual apărut pentru a crea o nouă bază de date.

Odată creată baza de date, urmează crearea efectivă a tabelelor componente. O metodă este prezentată în continuare. Se va crea un fişier text care să cuprindă descrierea unei comenzi SQL `CREATE TABLE`. De exemplu, acesta poate fi intitulat `studenti.sql`, conţinând următoarea secvenţă:

```
CREATE TABLE studenti (
  nr int not NULL auto_increment primary key,
        nume varchar(25) NULL,
        prenume varchar(25) NULL,
        nota int NULL
    );
```

Pentru a crea efectiv tabela, se foloseşte comanda `mysql studenti<studenti.sql` şi astfel se creează în directorul `\data`, subdirectorul `\studenti`, cu fişierele `studenti.myi`, `studenti.myd` şi `studenti.frm`.

8.7.2. Drivere JDBC pentru MySQL

Datorită faptului că serverul MySQL este *open source*, deci are sursele accesibile şi bine documentate, există mai multe implementări de drivere JDBC libere, disponibile pentru MySQL şi realizate de diferiţi dezvoltatori independenţi. Prezentăm o listă a driverelor existente în continuare:

Nume driver	Tip licenţă	Versiuni JDBC	Pagina de referinţă	Calea completă
Connector/J	*LGPL*	1.x şi 2.x	`http://www.mysql.com/Downloads/Connector-J/mysql-connector-java-2.0.14.zip`	`com.mysql.jdbc.Driver`
Mm (GNU)	*LGPL*	1.x şi 2.x	`http://mmmysql.sourceforge.net/`	`org.gjt.mm.mysql.Driver`
Twz	-	1.x	`http://www.voicenet.com/~zellert/tjFM/`	`twz1.jdbc.mysql.jdbcMysqlDriver`
Caucho	*QPL*	2.x	`http://www.caucho.com/projects/jdbc-mysql/index.xtp`	`com.caucho.jdbc.mysql.Driver`

Dintre acestea, *mmMySQL* (MM – provine de la Mark Matthews, autorul lui) este driverul recomandat în manualul MySQL, fiind considerat cel mai performant driver gratuit până la apariţia lui *Connector/J*, care nu este altceva decât o îmbunătăţire a acestuia. Driverul este împachetat într-un fişier `.jar` ce trebuie adăugat în `CLASSPATH`. Numele complet al driverului este `org.gjt.mm.mysql.Driver`, subprotocolul `mysql` şi formatul URL-ului JDBC:

```
    jdbc:mysql://[numegazda][:port]/numebazadate[?param1=val1]
    [&param2=val2]...
```

De exemplu, URL-ul pentru o bază de date numită `arhiva`, care este pe calculatorul local este `jdbc:mysql://localhost:arhiva`.

Odată cu numele maşinii, al portului şi al bazei de date, driverul mai permite setarea unor proprietăţi interne cu diverse valori introduse de utilizatori pe care le prezentăm în tabela următoare:

Nume proprietate	Mod de folosire	Valoare implicită
`User`	Utilizatorul care se conectează.	nici una
`Password`	Parola folosită pentru conectare.	nici una
`autoReconnect`	Va încerca să se reconecteze driverul în cazul în care conexiunea moare? (true/false)	false
`maxReconnects`	Dacă `autoReconnect` este setat, de câte ori va încerca driverul să se autoconecteze?	3
`initialTimeout`	Dacă `autoReconnect` este setat, timpul inițial de așteptare între reconectări (în secunde).	2
`maxRows`	Numărul maxim de linii care va fi returnat (0 înseamnă că se vor returna toate liniile).	0
`useUnicode`	Va folosi driverul setul de caractere Unicode atunci când va mânui șiruri de caractere? (true/false)	false
`characterEncoding`	Dacă `useUnicode` este true, ce tip de codare de caractere va folosi driverul?	nici una

Valorile proprietăților pot fi transmise în momentul realizării conexiunii, folosind diferitele suprascrieri ale metodei `getConnection()`, pe care clasa `DriverManager` le pune la dispoziție.

Suprascrierea metodei	Explicații
`public static Connection` **`getConnection`** `(String url)`	Permite realizarea unei conexiuni folosind un URL care poate să conțină și valori pentru proprietăți.
`public static Connection` **`getConnection`** `(String url, Properties info)`	Permite realizarea unei conexiuni folosind un URL și o listă de proprietăți specificate printr-un obiect `Properties`.
`public static Connection` **`getConnection`** `(String url, String utilizator, String parola)`	Permite realizarea unei conexiuni folosind un URL, un nume de utilizator și o parolă.

Prezentăm, în ordine, câte un exemplu pentru fiecare tip de conectare:

```
Connection con = DriverManager.getConnection(
"jdbc:mysql://localhost/arhiva?user=utilizator&password = parola&
useUnicode=true&characterEncoding=UTF8");

Properties p = new Properties();
p.put("user", "utilizator");
p.put("password", "parola");
p.put("useUnicode", "true");
p.put("characterEncoding", "UTF-8");
Connection con = DriverManager.getConnection(url, p);
Connection con = DriverManager.getConnection(
"jdbc:mysql://localhost/arhiva", "utilizator", "parola");
```

Trebuie remarcat faptul că, prin intermediul proprietăţilor `useUnicode` şi `characterEncoding`, se poate seta ca driverul să folosească codarea Unicode atunci când se lucrează cu şiruri de caractere. Prin această metodă se forţează folosirea diacriticelor. Acest lucru este extrem de important, deoarece, intern, serverul MySQL foloseşte implicit codarea *Latin1* (dacă nu se setează o alta), şi nu Unicode, aşa cum este cazul limbajului Java. De fapt, MySQL nici nu oferă suport pentru Unicode până la versiunea 4.0. Atunci când diacriticele sunt trimise spre serverul MySQL după ce au fost culese în componente grafice precum cele Swing, dacă driverul nu este setat pe Unicode, în urma interogărilor, vor fi readuse de pe server în mod eronat. Implicit, după cum se observă din tabela anterioară, driverul nu este setat să accepte Unicode. Pentru mai multe informaţii, puteţi consulta capitolul „Internaţionalizarea aplicaţiilor".

Una dintre soluţiile cele mai comode pentru gestionarea proprietăţilor ar fi folosirea fişierelor cu proprietăţi, gestionate prin clasa `ResourceBundle` sau prin clasa `Properties`. Presupunem că fişierul `resurse.properties` are drept conţinut:

```
Driver = org.gjt.mm.mysql.Driver
URL = jdbc:mysql://localhost/date?user=nume&password=parola
```

Prezentăm în continuare o metodă de a folosi acest fişier cu resurse, pentru a ne conecta la baza de date `date` şi a lista toate intrările din tabela `arhive`:

```
import java.sql.*;
import java.util.*;
public class Conectare {
  public static void main(String argv[]) {
    Connection con = null;
    ResourceBundle grup = ResourceBundle.getBundle(
        "resurse");
    try {
      String url = grup.getString("URL");
      // inregistram driverul
      Class.forName(grup.getString("Driver"));
      // facem conexiunea
      con = DriverManager.getConnection(url);
      Statement instructiune = conexiuneBazaDate.
         createStatement();
      // selecteaza toate inregistrarile din tabela arhive
      String sql = "select * from arhive";
      ResultSet rs = instructiune.executeQuery(sql);
      while (rs.next())  {
      System.out.println(rs.getString("nr_curent") + ", " + rs.
         getString("nume") + ", " + rs.getString("prenume"));
      }
```

```
            rs.close();
            instructiune.close();
            }
         catch( SQLException e ) {
            e.printStackTrace( );
         }
         finally {
            if( con != null ) {
            try { con.close( ); }
            catch( Exception e ) { }
            }
         }
      }
   }
```

Pentru driverul *Connector/J*, tot ceea ce a fost prezentat anterior rămâne valabil, cu observaţia că acum clasa care implementează interfaţa `java.sql.Driver` este `com.mysql.jdbc.Driver` şi se află într-o arhivă `.jar`, având numele de forma `mysqlconnectorjXXXbin.jar` (unde `XXX` este versiunea driverului), care trebuie inclus în `CLASSPATH`. Spre exemplu, pentru înregistrarea driverului *Connector/J* se va folosi un apel:

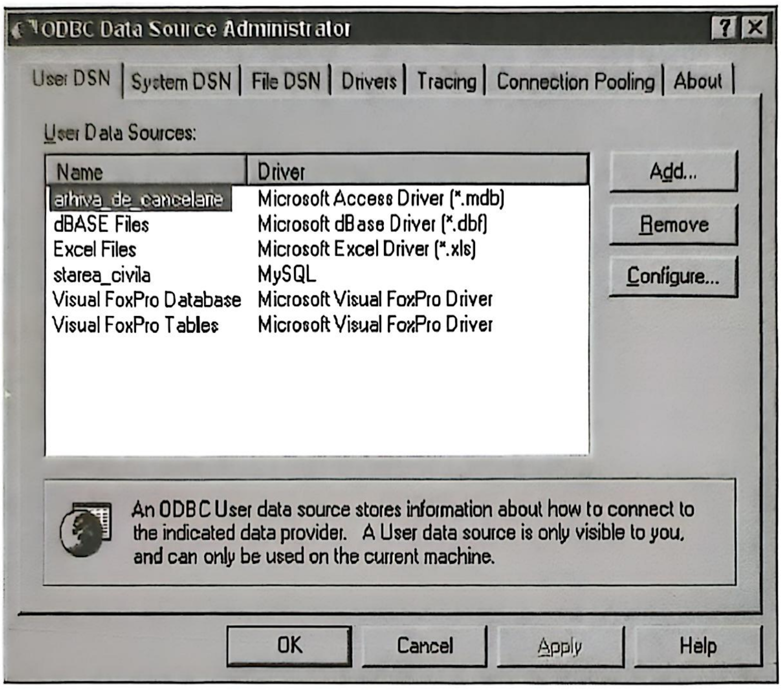
```
Class.forName("com.mysql.jdbc.Driver").newInstance();
```

O aplicaţie completă, care îmbină API-urile JDBC şi Swing pentru a realiza conectări la serverul de baze de date MySQL, puteţi găsi în secţiunea dedicată componentei grafice `JTable` din capitolul Swing.

8.8. Utilizarea punţii JDBC-ODBC

În cazul punţii JDBC-ODBC, timpul de acces la bazele de date este de câteva ori mai mare decât în cazul unui driver de tip 4 şi se aşteaptă ca această metodă să fie folosită doar experimental, atunci când nu avem încă la dispoziţie un alt tip de driver pentru DBMS-ul folosit. Puntea face legătura dintre aplicaţie şi ODBC, ce reprezintă un mecanism standard orientat spre programatorii de C/C++, pe care sistemul de operare Windows îl foloseşte ca interfaţă universală pentru conectarea cu baze de date de diverse tipuri. Motivele pentru care se foloseşte o punte, şi nu se apelează direct din aplicaţie API-ul ODBC folosind JNI, sunt multiple, cel mai important fiind independenţa totală faţă de facilităţile oferite de sistemul de operare peste care este instalată platforma Java. În ODBC, bazele de date sunt reprezentate ca surse de date, identificate printr-un nume (*DSN – Data Source Name*) şi accesibile printr-un driver ODBC corespunzător. Putem spune că, în cazul folosirii unui driver de tip 1, avem de-a face, de fapt, cu două drivere şi anume puntea propriu-zisă, reprezentată de driverul JDBC-ODBC, respectiv driverul ODBC corespunzător DBMS-ului interogat, care trebuie instalat în prealabil. Acest lucru poate deseori determina apariţia unor erori neaşteptate din cauza proastei interacţiuni dintre cele două drivere.

Prezentăm în continuare un exemplu ce ilustrează modul prin care se accesează dintr-o aplicaţie Java o bază de date creată în Access, sub sistemul de operare Windows 98. Presupunem că baza de date a fost deja creată şi are numele `arhiva.mdb`. Pentru a putea face interogări asupra ei, trebuie mai întâi să o introducem ca sursă de date în ODBC. Pentru aceasta, trebuie urmaţi următorii paşi: din `ControlPanel` alegem `ODBC Data Sources (32 bit)`, ceea ce determină apariţia ferestrei `ODBC Data Source Administrator`. În rubrica `User DSN`, folosind butonul `Add...` şi scenariul (engl. *wizard*) pe care acesta îl declanşează, selectăm, pe rând, driverul ODBC (în cazul nostru, spre Access), respectiv baza de date asupra căreia se doreşte să se facă interogările `arhiva.mdb`, şi punem numele acestei surse de date (engl. *Data Source Name*) `arhiva_de_cancelarie`, ca în exemplul următor:



Name	Driver
arhiva_de_cancelarie	Microsoft Access Driver (*.mdb)
dBASE Files	Microsoft dBase Driver (*.dbf)
Excel Files	Microsoft Excel Driver (*.xls)
starea_civila	MySQL
Visual FoxPro Database	Microsoft Visual FoxPro Driver
Visual FoxPro Tables	Microsoft Visual FoxPro Driver

An ODBC User data source stores information about how to connect to the indicated data provider. A User data source is only visible to you, and can only be used on the current machine.

Până în acest punct, am creat sursa de date. Pentru conectarea efectivă, trebuie să ataşăm aplicaţiei driverul punte spre ODBC. Driverul este inclus în distribuţia standard şi, în consecinţă nu mai este necesar să introducem în `CLASSPATH` calea spre clasele componente, ci este de ajuns să-l încărcăm dinamic printr-un apel `Class.forName("sun.jdbc.odbc.JdbcOdbcDriver")`. Urmează realizarea conexiunii, care se face cel mai convenabil printr-o secvenţă de forma:

```
// generarea adresei
String url="jdbc:odbc:arhiva_de_cancelarie";
// setarea proprietatilor
java.util.Properties prop = new java.util.Properties();
prop.put("charSet", "UTF-8");
prop.put("user", "utilizator");
prop.put("password", "parola");
```

```
        // conectarea la baza de date
        con = DriverManager.getConnection(url, prop);
```

Prin intermediul proprietății `charset`, se oferă suport pentru internaționalizare, fiind posibilă alegerea unui alt set de caractere, diferit de cel folosit implicit de DBMS. Această facilitate este valabilă numai pentru driverele JDBC-ODBC care vin începând cu distribuția 1.4 a platformei Java.

Prezentăm în continuare o metodă prin care se testează conexiunea la baza de date `arhiva.mdb`, presupunând că s-a creat mai întâi pentru aceasta sursa de date `arhiva_de_cancelarie`, așa cum am arătat într-un paragraf precedent:

```
static String driver = " sun.jdbc.odbc.JdbcOdbcDriver";
      static String parola = "";
      static String utilizator = "";
      static String subProtocol = "odbc";
      static String bazaDate = "arhiva_de_cancelarie";
//.....
public static boolean testeazaConexiune()
  {
   boolean rezultat = true;
   try {
      Class.forName(driver); // driverul este incarcat dinamic
       String url = "jdbc:" + subProtocol + ":" + bazaDate;
       System.out.println("A rezultat urmatorul url: " + url);
       //se va afisa: A rezultat urmatorul url: jdbc:
       //odbc:arhiva_de_cancelarie
       Connection conex = DriverManager.getConnection
          (url,utilizator,parola);
       Statement stare = conex.createStatement();
   }
   catch(Exception ex) {
       System.err.println("Eroare la conectare: " +
       ex.toString());
       rezultat = false;
   }
   return rezultat;
  }
```

Pentru a testa viteza de acces, putem instala driverul ODBC cu numele MyODBC (denumit oficial *ODBC Connector*), care face legătura dintre ODBC și serverul MySql, driver pe care îl puteți descărca gratuit de la adresa: *www.mysql.com/products/myodbc*. Acest driver vă permite, de asemenea, să folosiți un *Add-in* pentru Access, având numele *MyAccess*, pe care îl găsiți gratuit la adresa *www.mysql.com/portal/software/item-242.html* și cu care puteți face conversii ale bazelor de date Access în format MySql.

Codul Java folosit în aplicaţii rămâne valid după tranziţia spre alt sistem de gestiune a bazelor de date, deoarece orice driver de orice tip respectă API-ul JDBC, singurul pas necesar tranziţiei fiind schimbarea driverului.

8.9. Tranzacţii

O tranzacţie constă în una sau mai multe instrucţiuni SQL, care sunt executate complet, într-o formă atomică, iar toate modificările apărute în baza de date accesată sunt salvate permanent folosind metoda `commit()` sau anulate printr-o metodă `rollback()`. De asemenea, orice blocare a datelor conţinute în baza de date utilizată de acea tranzacţie este anulată prin apelarea uneia dintre metodele anterioare. După ce una dintre cele două comenzi `commit()` sau `rollback()` este executată, tranzacţia curentă se termină şi o alta poate începe.

Implicit, un obiect `Connection` creat este în modul *autocommit*, ceea ce înseamnă că atunci când o interogare este terminată, metoda `commit()` va fi apelată automat pentru a face permanente modificările apărute. În acest caz, interogarea efectuată se consideră ca fiind o tranzacţie de sine stătătoare. Dacă modul *autocommit* este dezactivat prin intermediul unei metode `setAutoCommit(false)`, tranziţia nou începută nu se va termina decât printr-un apel explicit al uneia dintre metodele `commit()` sau `rollback()` şi va îngloba toate instrucţiunile SQL executate între timp. Operaţia de salvare permanentă sau de anulare are efect asupra tuturor interogărilor din tranzacţie. Dacă una dintre instrucţiunile SQL nu se execută cu succes, se va executa explicit `roollback()`, ceea ce înseamnă că toate modificările făcute de acele interogări sunt anulate, ajungându-se la starea de dinaintea începerii tranzacţiei. Altfel, toate modificările determinate de instrucţiunile SQL care compun tranzacţia sunt salvate printr-un apel explicit `commit()`. Acest lucru se întâmplă ca efect al proprietăţii de atomicitate a tranzacţiilor (a se vedea subsecţiunile următoare).

După cum se observă, începerea unei tranzacţii nu necesită nici un apel explicit, fiind implicit iniţiată în momentul terminării unei alte tranzacţii sau prin apelul `setAutoCommit(false)`.

Orice driver JDBC trebuie să suporte tranzacţiile. În fapt, pentru ca un driver să poată fi acceptat ca fiind compatibil JDBC, trebuie să ofere suport pentru tranzacţii.

8.9.1. Motivaţii pentru folosirea tranzacţiilor

Enumerăm câteva motive pentru care folosirea tranzacţiilor îşi găseşte justificare:

Necesitatea de a realiza operaţii atomice – presupune situaţia în care se doreşte ca mai multe operaţii separate să se execute împreună, într-o manieră atomică. Această abordare este utilă, spre exemplu, în cazul în care dorim să facem modificări asupra mai multor tabele aflate, eventual, în baze de date diferite pe maşina locală sau în reţea, modificări care îşi au rostul doar realizate împreună. În particular, putem da drept exemplu cazul transferului de bani dintr-un cont în altul, care presupune ca, odată suma de bani citită şi modificată din contul-sursă, să fie scrisă şi în contul-destinaţie. Dacă operaţia nu decurge în mod atomic, se va întâmpla, spre exemplu, ca suma să nu fie transferată efectiv şi să se regăsească şi în sursă, şi în destinaţie, ceea ce reprezintă un transfer incorect (inconsistenţă). Tranzacţiile rezolvă această problemă prin garantarea principiului „totul sau nimic".

Căderi ale maşinii sau ale reţelei – în cazul în care bazele de date sunt distribuite în reţea, situaţiile ce pot determina excepţii cresc. De asemenea, maşina pe care se află bazele de date poate să cadă la un moment dat. La fel, serverul de baze de date. În cazul în care căderea se petrece atunci când se face o scriere în baza de date, aceasta poate să fie deteriorată irecuperabil. Tranzacţiile sunt cele care oferă suport pentru recuperarea datelor.

Partajarea aceloraşi date de către mai mulţi utilizatori – într-un intranet apare frecvent posibilitatea accesării aceleiaşi baze de date de către mai mulţi clienţi în acelaşi timp. Conexiunile concurente pot determina o serie de erori. Amintim câteva dintre ele.

Citiri murdare (engl. *dirty reads*) – această problemă apare în cazul în care un client citeşte date dintr-o bază de date care nu a fost salvată definitiv pe suportul de stocare de un alt client prin operaţia `commit()`. Drept exemplu, presupunem următoarea succesiune de paşi:

1. Clientul C1 citeşte întregul X din baza de date. Presupunem că iniţial X este 0.
2. Clientul C1 adună 10 la X şi astfel baza de date are acum X = 10. Presupunem că C1 nu a apelat încă operaţia `commit()`, care ar fi determinat salvarea definitivă a bazei de date pe suport.
3. Un alt client C2 citeşte întregul X din baza de date. Valoarea pe care o citeşte din X este 10.
4. Clientul C1 renunţă (engl. *abort*) la tranzacţie, ceea ce determină ca X din tabelă să devină 0.
5. Clientul C2 adună 20 la X şi îl salvează în baza de date. Acum baza de date va conţine X = 30.

După cum se poate observa, clientul C2 foloseşte la pasul 5 o valoare a lui X care nu mai este valabilă din cauza pasului 4. Deci o aplicaţie a folosit o dată a tabelei înainte ca ea să fie stocată permanent, ceea ce determină o eroare în logica aplicaţiei.

Citiri nerepetabile (engl. *unreapeatable reads*) – această problemă apare atunci când un client citeşte date dintr-o tabelă, dar între timp acestea sunt modificate de un alt client. Ca succesiune de paşi:

1. Clientul C1 citeşte întregul X din baza de date.
2. Clientul C2 suprascrie întregul din tabelă cu o nouă valoare.
3. Clientul C1 citeşte iarăşi întregul X. Surprinzător, acesta nu mai are aceeaşi valoare.

Problema fantomelor (engl. *phantoms*) – această problemă presupune apariţia unei noi mulţimi de date între două operaţii de citire făcute de un acelaşi client. Pe paşi, problema poate fi privită astfel:

1. Clientul C1 interoghează aplicaţia şi obţine o mulţime de date rezultat.
2. Clientul C2 inserează noi date în baza de date.
3. Clientul C1 reapelează aceeaşi interogare, dar, surprinzător, rezultatul obţinut este altul.

Diferenţa între citiri nerepetabile şi problema fantomelor este aceea că în primul caz este vorba de modificări de date existente, iar în al doilea caz este vorba de date cum ar fi înregistrări sau coloane noi, care nu existau anterior.

8.9.2. ACID

În cazul folosirii tranzacţiilor, un număr de patru principii sunt întotdeauna garantate pentru operaţiile efectuate. Acestea poartă şi numele de proprietăţile ACID ale tranzacţiilor, denumire care vine de la abrevierea celor patru nume de proprietăţi: atomicitate (engl. *Atomicity*), consistenţă (engl. *Consistency*), izolare (engl. *Isolation*) şi durabilitate (engl. *Durability*).

Prezentăm în continuare descrierea celor patru proprietăţi.

Atomicitate – garantează că mai multe operaţii sunt grupate împreună şi apar ca o singură unitate continuă de lucru. De asemenea, atomicitatea garantează că mai multe operaţii sunt executate toate odată sau nici una.

Consistenţă – garantează că starea sistemului rămâne consistentă după ce o tranzacţie este terminată. Prin stare consistentă se înţelege o stare în care sistemul respectă una sau mai multe reguli. Sistemul poate trece şi prin stări temporare de inconsistenţă. Dar, în final, tranzacţiile garantează consistenţa sistemului.

Izolare – permite ca mai mulţi clienţi să citească şi să scrie într-o bază de date fără să ştie unul de altul, deoarece tranzacţiile sunt izolate una de alta. Este metoda prin care se rezolvă problemele legate de partajarea resurselor (a se vedea subsecţiunile anterioare). Acest lucru este util pentru cazul mai multor clienţi care accesează baza de date simultan, fiecare având impresia că el este singurul conectat. Tranzacţiile asigură izolarea prin intermediul sincronizărilor la nivelul inferior al DBMS, prin blocări (engl. *locks*) automate ale datelor sensibile, care asigură că odată ce un client accesează zona sensibilă (critică), nimeni altcineva nu mai poate face acest lucru. Încă o dată trebuie remarcat faptul că, prin folosirea tranzacţiilor, programatorii nu mai trebuie să scrie cod pentru tratarea concurenţelor multiple, deoarece de acest aspect se ocupă DBMS-ul. Tot ce trebuie să facem este să specificăm tipul de acces.

Durabilitate – garantează că toate actualizările permanent salvate folosind `commit()` supravieţuiesc căderilor de sistem, erorilor de reţea, căderii hard diskului sau căderilor de tensiune. Acest lucru este posibil datorită faptului că tranzacţiile păstrează un fişier de siguranţă în care sunt înregistrate toate operaţiile asupra bazelor de date, înainte ca acestea să fie efectiv comise. Datele salvate permanent pot fi astfel reconstituite prin reaplicarea paşilor salvaţi în acest fişier.

Vom vorbi în continuare despre izolarea tranzacţiilor, care necesită o abordare specială, din cauza problemelor ce pot apărea în momentul partajării aceloraşi resurse de către mai mulţi clienţi.

8.9.3. Niveluri de izolare

Dacă un DBMS suportă tranzacţii concurente, programatorul aplicaţiei care se conectează la acel DBMS poate specifica nivelul de izolare a tranzacţiilor efectuate. Izolarea poate fi strictă sau relaxată. *O izolare strictă* determină separarea totală a tranzacţiilor, care se realizează prin restricţii impuse asupra datelor din tabelele bazei, soldate cu o scădere a performanţelor. De aceea trebuie ales cu grijă nivelul de izolare de care este nevoie. Există mai multe niveluri de izolare, descrise prin cinci constante din clasa `Connection`, pe care le vom prezenta în tabelul care urmează:

Nivel izolare	Explicaţii
`TRANSACTION_NONE`	Nu sunt permise tranzacţiile.
`TRANSACTION_READ_UNCOMMITTED`	Nu oferă nici un fel de izolare, dar garantează performanţe înalte în ceea ce priveşte viteza de acces. Nu este indicat să se folosească acest nivel de izolare în sisteme cu conexiuni concurente şi, mai ales, în cele cu date sensibile, cum ar fi sistemul bancar, din cauza problemelor inevitabile care vor apărea.
`TRANSACTION_READ_COMMITTED`	Rezolvă problema citirilor murdare. Folosind acest nivel de izolare, nu se vor citi datele care au fost modificate în tabela accesată, fără a fi salvate definitiv prin metoda `commit()`. `Read_committed` este nivelul implicit de izolare pentru cele mai multe servere printre care `Oracle` şi `MS SQL`.
`TRANSACTION_REPEATABLE_READ`	Rezolvă problemele anterioare şi problema citirilor nerepetabile.
`TRANSACTION_SERIALIZABLE`	Rezolvă problemele anterioare şi problema fantomei. Folosind acest nivel, tranzacţiile sunt izolate total, ceea ce înseamnă că ele se vor executa secvenţial, independent una faţă de alta. Alegerea acestui nivel va determina încetinirea simţitoare a tranzacţiilor concurente.

Atunci când un obiect `Connection` este creat, nivelul de izolare al conexiunii depinde de driver. Programatorii pot face apeluri de forma `setTransactionIsolation (TRANSACTION_REPEATABLE_READ)` şi un nou nivel de izolare va fi valabil pe tot parcursul conexiunii. Schimbarea modului de izolare a tranzacţiilor în timp ce acestea se desfăşoară nu este recomandată, deoarece determină execuţia imediată a metodei `commit()`, care va face permanente modificările din tabelă.

8.10. Concluzii

Acest capitol prezintă aspectele fundamentale legate de conectarea aplicaţiilor Java la sistemele de gestiune a bazelor de date (DBMS). Unele concepte, precum sursele de date şi procedurile stocate, nu sunt discutate în acest capitol.

Pentru început au fost descrise aspecte legate de API-ul JDBC şi specificaţiile care stau la baza acestui API. Urmează clasificarea driverelor JDBC şi adresele paginilor Web de unde se pot afla mai multe informaţii şi de unde, eventual, acestea se pot descărca. Mai apoi, este pus la dispoziţia cititorilor un scurt rezumat al interogărilor SQL, dintre care unele sunt folosite în exemplele capitolului. Urmează maparea dintre tipurile SQL şi tipurile Java, respectiv informaţii despre ţipurile SQL3 suportate de JDBC 2.0.

Este prezentată pe larg accesarea unei baze de date, urmată de prezentarea DBMS-ului MySQL, cu două dintre driverele cele mai performante, şi a punţii JDBC-ODBC. În final, se face o incursiune în tranzacţii şi sunt prezentate în special o serie de probleme determinate de conexiuni multiple, precum şi metodele de rezolvare a lor.

Ca o concluzie, putem afirma că JDBC reprezintă unul dintre cele mai mature API-uri independente pentru accesarea bazelor de date şi că se bucură de sprijinul unor giganţi informatici (precum Oracle), care şi-au fundamentat produsele lor pe tehnologii Java.

8.11. Test-grilă

Întrebarea 8.11.1. Ce reprezintă un sistem de gestiune al bazelor de date (DBMS)?

a) O aplicaţie care accesează baze de date ţinute de un server.
b) Un sistem de reguli şi convenţii care reglementează schimbul de date între componenţii unei arhitecturi client-server.
c) Un sistem care permite gestionarea bazelor de date şi oferă modalităţi ca acestea să fie accesate de clienţi via conexiuni.
d) Un gestionar de conexiuni între un client şi un server de baze de date.

Întrebarea 8.11.2. Ce reprezintă un driver JDBC?

a) O componentă fizică ce trebuie instalată ca un modul al calculatorului, pentru a ne putea conecta la un server de baze de date.
b) O componentă software care intermediază între un client Java şi un DBMS (SGBD).
c) O aplicaţie server care trimite date clienţilor în urma unor interogări SQL.

Întrebarea 8.11.3. Ce reglementează specificaţia JDBC?

a) Cum trebuie construiţi clienţii care folosesc driverele JDBC.
b) O serie de interfeţe standard ce trebuie suprascrise de creatorii de drivere JDBC.
c) Sintaxa SQL.

d) Versiunile driverelor JDBC.
e) Concordanţa dintre JDBC şi ODBC.

Întrebarea 8.11.4. Fie următorul apel:

```
Statement declar = con.createStatement(
   ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.
   CONCUR_UPDATABLE);
```

Ce proprietăţi vor avea instanţele `ResultSet` obţinute folosind acest obiect `Statement`?

a) Vor avea cursor deplasabil numai înainte şi vor fi senzitive la modificările din tabelele interogate.
b) Vor avea cursor poziţionabil pe orice linie, permiţând modificarea programatică a tabelei interogate.
c) Vor fi insenzitive la modificările survenite în tabele, permiţând modificarea programatică a tabelelor interogate.

Întrebarea 8.11.5. Fie tabela `t` cu 10 înregistrări şi având coloanele `a` şi `b`, respectiv următoarea secvenţă de cod:

```
ResultSet rez = instructiune.executeQuery("select * from t");
rez.last();
rez.updateInt("a", 100);
rez.updateString(2,"I");
rez.first();
```

Care dintre afirmaţiile de mai jos sunt adevărate după execuţia codului?

a) Se va adăuga în tabela `t` o nouă înregistrare.
b) Se va modifica ultima linie din tabelă pentru a actualiza elementul de pe coloana `a` la `100` şi elementul de pe coloana `b` la `"I"`.
c) Tabela va rămâne nemodificată.

Întrebarea 8.11.6. Ce reprezintă un obiect `java.sql.Clob` obţinut în urma unui apel `getClob` al clasei `ResultSet`?

a) Un şir de caractere cu o lungime mai mare decât 256.
b) O referinţă spre un obiect de tip SQL `BLOB` stocat de DBMS.
c) Un tip de date SQL definit de utilizator.

Întrebarea 8.11.7. Fie secvenţa de cod:

```
Class.forName("com.mysql.jdbc.Driver");
Connection con = DriverManager.getConnection(
"jdbc:mysql://localhost/arhiva?user=utilizator&password=parola&
useUnicode=true&characterEncoding=UTF-8");
```

Care dintre afirmaţiile următoare este adevărată?

a) Codul permite obţinerea unei conexiuni la serverul MySQL folosind driverul MMMySQL.

b) Codul permite realizarea unei conexiuni la serverul MySQL şi cere driverului să folosească standardul `Unicode` în comunicaţie.

c) Codul înregistrează driverul *Connector/J* şi realizează o conexiune la serverul MySQL folosind acest driver.

Întrebarea 8.11.8. Ce reprezintă ODBC?

a) Implementarea obiectuală a driverelor JDBC.

b) Un driver JDBC care permite conectarea la baze de date Access.

c) O sursă de date.

d) Un server DBMS.

Întrebarea 8.11.9. Ce determină un apel `commit()`?

a) Terminarea unei tranzacţii.

b) Începerea unei noi tranzacţii.

c) Anularea unei tranzacţii.

Întrebarea 8.11.10. Care dintre afirmaţiile următoare reprezintă problema citirii murdare (engl. *dirty read*)?

a) Un client citeşte date dintr-o tabelă, dar între timp alt client modifică respectivele date. Recitind, primul client obţine alte informaţii.

b) Un client scrie o informaţie în baza de date fără a salva definitiv datele şi un alt client foloseşte între timp informaţia. Primul client renunţă, iar al doilea modifică valoarea citită şi o salvează.

c) Un client aplică metoda `commit`, în timp ce altul aplică metoda `rollback`.

8.12. Exerciţii propuse spre implementare

Exerciţiul 8.12.1. Să se realizeze o aplicaţie care creează tipuri SQL utilizator, încapsulând clase Java deja existente, permiţând astfel stocarea obiectelor instanţe ale respectivelor claselor Java în baze de date gestionate de un DBMS oarecare.

Exerciţiul 8.12.2. Să se implementeze o aplicaţie care oferă toate proprietăţile şi informaţiile despre un DBMS la care aceasta poate crea o conexiune folosind un driver JDBC. Se va folosi clasa `DatabaseMetaData`.

Exerciţiul 8.12.3. Scrieţi o clasă Java care testează posibilitatea realizării de conexiuni spre diverse DBMS-uri, folosind diverse drivere.

Exerciţiul 8.12.4. Să se creeze o aplicaţie care face conversia bazelor de date dinspre un DBMS spre un altul. În particular, să se implementeze posibilitatea de a face conversii de baze de date dinspre MySQL spre Access şi viceversa.

Exerciţiul 8.12.5. Testaţi diferenţele în viteza de execuţie pentru interogările executate asupra serverului de baze de date MySQL, folosind toate driverele gratuite existente.

Exerciţiul 8.12.6. Testaţi pe DBMS-ul preferat problemele care pot apărea din cauza neizolării tranzacţiilor. Aplicaţi nivelurile de izolare pentru a observa noii timpi de acces spre DBMS.

Exerciţiul 8.12.7. Realizaţi o metodă de salvare a interogărilor SQL efectuate asupra unui DBMS într-un fişier de siguranţă XML, înainte ca acestea să fie trimise spre server şi executate. Realizaţi o modalitate de apelare pas cu pas a interogărilor salvate în acel fişier, pentru a recupera o tabelă dacă aceasta a fost distrusă întâmplător.

8.13. Proiecte propuse spre implementare

Proiectul 8.13.1. Realizaţi o aplicaţie Java care să administreze bazele de date ţinute de un DBMS. Aplicaţia va arăta bazele de date, va permite crearea de noi baze de date şi tabele, actualizări asupra tabelelor componente etc. DBMS-ul va putea fi ales în timpul rulării aplicaţiei administrator, iar driverul va putea fi descărcat de pe internet şi încărcat în aplicaţie în timpul rulării acesteia.

Proiectul 8.13.2. Creaţi o aplicaţie Java care să salveze obiectele Java în tabele prin asocierea câmpurilor din obiecte la câmpurile tabelelor. Perfecţionaţi acest procedeu pentru a reprezenta o alternativă la serializarea obiectelor Java.

Proiectul 8.13.3. Creaţi o aplicaţie Java care să salveze în fişiere XML datele ţinute de un DBMS. Aceasta va reprezenta o modalitate de a realiza copii de siguranţă ale tabelelor. Să se asigure o metodă de restaurare a tabelelor a căror salvare s-a realizat.

Proiectul 8.13.4. Creaţi un adaptor universal care să permită vizualizarea într-un `JTabel` a unui `ResultSet`, rezultat în urma unei interogări SQL. Vor varia numele bazei de date, driverul şi interogarea SQL. A se vedea exemplul prezentat în capitolul Swing la secţiunea dedicată componentei grafice `JTable`. Tabela va putea fi actualizată direct din interfaţa grafică. Se vor folosi şi avantajele oferite de drivere conforme specificaţiei 2.0 sau 3.0.

Proiectul 8.13.5. Realizaţi o aplicaţie care să testeze viteza de accesare a datelor gestionate de mai multe DBMS-uri. Aplicaţia se va putea conecta la mai multe baze de date, aflate eventual pe mai multe DBMS-uri, şi va permite realizarea de comparaţii între diverşii timpi de accesare obţinuţi.

Proiectul 8.13.6. Realizaţi o aplicaţie pentru controlul versiunilor (CVS) care să stocheze atât surse Java ca tipuri `CLOB`, cât şi clase ca tipuri `BLOB` în baze de date şi să le furnizeze la cerere clienţilor. În cazul în care baza de date curent interogată nu răspunde, se va oferi posibilitatea conectării la o alta, eventual gestionată de un alt DBMS.

Proiectul 8.13.7. Realizaţi o aplicaţie (sau un applet) Java care să permită celor care o accesează să rezolve teste-grilă şi să primească punctajul obţinut. Testarea se va face contra cronometru, cu posibilitatea creşterii dificultăţii la cererea clientului. Testele vor fi împărţite pe domenii şi vor fi stocate în baze de date.

9. Programarea reţelelor

9.1. Cuvinte-cheie

- canal de comunicaţie (engl. *socket*), datagramă, client/server
- TCP/IP, UDP/IP
- adrese IP, URL
- apelul metodelor la distanţă (engl. *Remote Method Invocation*)
- programare distribuită

9.2. Introducere

Programarea rețelelor de calculatoare este de obicei descrisă folosind conceptele de *client* și *server*. Un *server* este o aplicație care rulează pe un calculator gazdă (*host*) ce furnizează o conexiune și informații utile din momentul stabilirii acelei conexiuni. Uneori, prin extrapolare, se utilizează termenul de server atât pentru aplicația care rulează, cât și pentru calculatorul gazdă pe care rulează aplicația. Un *client* este o aplicație care rulează pe un calculator din rețea, căutând să stabilească o conexiune cu un server. De regulă, se pot conecta mai mulți clienți simultan la un server.

Serverul este un furnizor de informații, iar clientul este un consumator. Clientul are nevoie de informații, se conectează la un server căruia îi adresează o cerere. Respectivul server trimite înapoi un răspuns care poate conține informația cerută sau indică faptul că nu posedă respectiva informație. Clientul poate să continue emiterea de cereri, eventual altor servere.

Pentru Web, navigatoarele sunt clienți, iar serverele Web sunt servere. De exemplu, dacă indicăm navigatorului adresa *http://www.infoiasi.ro/index.html*, atunci acesta va adresa o cerere serverului *www.infoiasi.ro* pentru documentul *index.html*. Dacă serverul nu posedă respectiva pagină, atunci va returna un cod de eroare. În caz contrar, va trimite pagina Web cerută. Pentru vizualizarea documentului Web, navigatorul poate avea nevoie și de alte documente. În acest caz, va trebui să mai adreseze câteva cereri serverului Web. Eventual, dacă pagina conține o imagine aflată pe alt server, navigatorul va adresa o cerere și respectivului server.

Pentru a se putea realiza o comunicare între clienți și server, aceștia vor trebui să respecte un set de reguli (*protocol*). De exemplu, clientul trimite mai întâi cererea, după care i se trimite un răspuns. Mesajele trimise trebuie să fie înțelese de cei doi parteneri.

9.2.1. Adrese, porturi și socketuri

Pentru a comunica între ele, calculatoarele trebuie să fie conectate fizic. Cea mai mare rețea cunoscută este internetul, care este, de fapt, o rețea de rețele de calculatoare.

Un calculator este unic identificat într-o rețea prin adresa IP (engl. *Internet Protocol*). Aceasta este formată din patru numere cuprinse între 0 și 255, separate prin simbolul punct (de exemplu: `193.231.30.255`). Identificarea unui calculator din rețea se mai poate realiza prin numele asociat. De exemplu, `thor.info.uaic.ro` este numele calculatorului cu adresă IP mai sus menționată. Aceste nume se mai numesc și *domenii*. Domeniile sunt mult mai ușor de reținut.

Numele de domeniu denotă o mașină specifică unui număr de niveluri. Nivelurile se precizează, în ordine, de la general la specific, de la dreapta la stânga. De exemplu, adresa `thor.info.uaic.ro` semnifică mașina `thor`, din domeniul Facultății de Informatică, din cadrul Universității „Al.I. Cuza", România.

Pentru a schimba date, clienții și serverele au nevoie de mai mult decât adrese. De exemplu, multe servere pot rula simultan pe aceeași mașină, deci au aceeași adresă IP.

Într-un calculator, mecanismul utilizat pentru stabilirea unei „întâlniri" se numește *port*. Porturile sunt numere întregi cuprinse între 0 și 65535. De regulă, valorile mai mici de 1024 sunt rezervate pentru servicii predefinite (de exemplu: poșta electronică, *ftp* – transfer de fișiere, *http* – transfer de documente hipertext). Programatorii pot utiliza pentru propriile aplicații porturile mai mari de 1024.

Combinația dintre adresa IP și numărul de port este întrebuințată pentru crearea unui termen abstract, numit *socket*. Acesta se mai numește în literatura de specialitate *canal de comunicație* sau, simplu, *conexiune*. Un client caută serverul, stabilește o conexiune prin care informația poate fi trimisă, iar la final se va deconecta lăsând liber socketul pentru o utilizare ulterioară. Un socket furnizează facilități pentru crearea de fluxuri de intrare și ieșire, care permit datelor să fie schimbate între client și server.

Pentru un socket, numărul de port poate fi specificat prin program sau asignat de către sistemul de operare. Când un socket trimite un pachet, acestuia trebuie să-i adăugăm informații despre destinația pachetului, cum ar fi:

- adresa mașinii din rețea care trebuie să primească pachetul;
- port pentru indicarea socketului corespunzător care va primi datele.

Atunci când se stabilește o conexiune, atât clientul, cât și serverul, vor avea câte un socket. Comunicarea efectivă se va realiza între socketuri.

În mod necesar, socketul serverului va indica la creare numărul portului, după care va aștepta realizarea de conexiuni cu clienții. Numerele de port pentru socketurile serverului sunt numere cunoscute pentru programele-client. De exemplu, un server FTP (*File Transfer Protocol*) utilizează un socket care „ascultă" la portul 21. Dacă un program-client dorește să comunice cu un server FTP, acesta va ști să contacteze socketul care ascultă la portul 21.

De obicei, sistemul de operare stabilește numărul portului pentru socketurile-client. Alegerea portului pentru client nu prezintă importanță, întrucât serverul nu trebuie să știe dinainte de pe care port vin clienții. Când un socket-client trimite un pachet către un socket-server, pachetul conține și numărul de port și adresa mașinii clientului. Astfel, cu ajutorul informațiilor primite, serverul este capabil să știe cine a adresat cererea. Aceste date sunt utilizate pentru a răspunde clientului.

Când se folosesc socketuri, trebuie să decidem ce tip de protocol dorim să utilizăm pentru transportul datelor în rețea:

- protocol orientat pe conexiune;
- protocol neorientat pe conexiune.

În cazul protocolului orientat pe conexiune, un client stabilește o conexiune cu un server via socketuri. Odată stabilită conexiunea, un protocol orientat pe conexiune asigură că datele s-au trimis sigur, adică:

- fiecare pachet trimis este și primit. De fiecare dată când printr-un socket se trimite un pachet, acesta așteaptă o confirmare că pachetul s-a primit cu succes. Dacă nu se primește confirmarea într-un timp precizat, se trimite din nou pachetul. Se continuă să se trimită pachete până când transmisia are succes sau până când se decide că transmisia este imposibilă;

- pachetele sunt citite de socketul-destinaţie în aceeaşi ordine în care au fost trimise. Datorită modului în care funcţionează reţelele, pachetele pot ajunge într-o ordine diferită faţă de cea în care au fost trimise. Protocolul orientat pe conexiune va permite socketului-destinaţie să restabilească ordinea în care au fost trimise pachetele. În comunicarea orientată conexiune este necesară participarea simultană a celor două aplicaţii.

Un protocol neorientat pe conexiune permite cea mai rapidă trimitere a pachetelor. Acesta nu garantează că pachetele care sunt trimise sunt citite în aceeaşi ordine de programul destinatar. Mai mult, programul destinatar nu trebuie neapărat să fie disponibil pentru recepţionarea datelor. Expeditorul trimite informaţiile, iar destinatarul le va citi eventual mai târziu, după care va răspunde. În schimb, este garantată primirea pachetelor (acestea nu vor fi pierdute). Iată două situaţii în care protocoalele neorientate pe conexiune sunt frecvent preferate celor orientate pe conexiune:

- când trebuie trimis doar un singur pachet şi garantarea trimiterii nu este crucială, atunci un protocol neorientat pe conexiune elimină timpul pierdut pentru crearea şi distrugerea unei conexiuni. De exemplu, protocolul orientat pe conexiune TCP/IP (*Transmission Control Protocol/Internet Protocol*) utilizează şapte pachete pentru trimiterea unui singur pachet, în timp ce protocolul neorientat pe conexiune UDP/IP (*User Datagram Protocol/Internet Protocol*) foloseşte doar unul. Un protocol pentru obţinerea timpului curent, de regulă, recurge la un protocol neorientat pe conexiune pentru a minimiza întârzierile;
- pentru aplicaţii în timp real, cum ar fi aplicaţiile audio/video, garantarea unei transmisii sigure nu este un avantaj. Aşteptările unor date care întârzie pot implica pauze importante în aplicaţia respectivă. Tehnicile de trimitere audio/video care utilizează un protocol neorientat pe conexiune au fost dezvoltate şi funcţionează mai bine. De exemplu, aplicaţia *RealAudio* utilizează un protocol neorientat pe conexiune pentru transmiterea sunetelor prin reţea. De asemenea, protocolul RTP (*Real Time Protocol*) este tot neorientat conexiune.

Diferenţele dintre conexiunile TCP şi UDP pot fi descrise prin următoarea analogie. TCP este asemenea unui apel telefonic: numărul de telefon este format o dată, apoi se stabileşte o conexiune care este valabilă până când legătura este întreruptă de către unul dintre participanţi. Pe de altă parte, UDP este echivalent cu sistemul de poştă obişnuită. Fiecare scrisoare (mesaj) este pusă într-un plic (pachet) separat şi vor ajunge separat sau nu la destinaţie. Pachetele pot fi pierdute şi nu este sigur dacă acestea vor fi recepţionate în ordinea în care au fost trimise. Cu toate acestea, dacă un pachet ajunge, suntem siguri că acesta este intact! Ambele protocoale, TCP şi UDP, transmit datele prin intermediul IP-ului, împărţind mesajul în pachete IP individuale.

9.3. Programarea reţelelor prin intermediul conexiunilor

Pachetul `java.net` furnizează două mecanisme de bază pentru accesarea datelor şi a resurselor prin intermediul unei reţele:

- *socketul*, care este mecanismul fundamental. Un socket permite programelor să schimbe grupuri de octeţi (pachete). Există clase în pachetul `java.net` care oferă suport pentru lucrul cu socketuri: clasele `Socket` şi `ServerSocket` sunt utile pentru realizarea unui transfer orientat conexiune, iar `DatagramSocket`, `DatagramPacket` şi `MulticastSocket` pentru cele neorientate conexiune.
- *URL* (*Uniform Resource Locator*), care este mecanismul de nivel înalt. Pachetul `java.net` conţine clasa `URL` pentru accesarea şi obţinerea unei resurse dintr-o reţea.

Implementarea socketurilor în Java este bazată pe biblioteca `Socket` care a fost parte originală din BSD UNIX. De aceea, există foarte multe similarităţi cu socketurile din alte limbaje de programare, indiferent de platforma acestora.

Un socket TCP pentru client corespunde unei instanţe a clasei `Socket`, iar pentru un server, unei clase `ServerSocket`. Odată stabilită conexiunea, se pot trimite sau primi fluxuri de la aplicaţia de la distanţă, comunicarea realizându-se prin intermediul fluxurilor. Se pot utiliza fluxurile de octeţi de tip `Filter` pentru adăugarea funcţiilor de nivel înalt pentru canalul de comunicare.

9.3.1. Clasa `ServerSocket`

Pentru a implementa un server orientat conexiune, trebuie să urmăm paşii:

- se construieşte o instanţă a clasei `ServerSocket` pentru a crea un socket ce va aştepta la un port precizat, cu scopul stabilirii conexiunilor;
- când se va accepta o conexiune (prin apelul metodei `accept()`), se va crea un obiect `Socket` care încapsulează conexiunea;
- se va utiliza obiectul `Socket` creat anterior pentru a obţine un flux de intrare de tip `InputStream`, respectiv unul de ieşire `OutputStream`, în vederea citirii şi scrierii informaţiilor către, respectiv de la client. Comunicarea cu clientul se va realiza deci prin intermediul fluxurilor;
- opţional, se vor putea crea noi fire de execuţie pentru fiecare conexiune nouă, astfel încât să se poată accepta în continuare noi conexiuni în timp ce sunt deservite celelalte cereri.

Pentru construirea obiectelelor `ServerSocket` trebuie să alegem un port pe maşina locală, cuprins între 1 şi 65535. Dacă se selectează numărul de port 0, atunci sistemul de operare va atribui un port liber şi valid de fiecare dată când rulează aplicaţia. Numărul de port ales trebuie apoi comunicat clienţilor pentru a se putea conecta la server.

După ce a fost creat un obiect `ServerSocket` se vor putea accepta conexiuni cu clienţii. Dacă apar mai multe cereri de conexiune din partea clienţilor, acestea vor fi depuse într-o coadă de aşteptare şi eliminate una câte una de îndată ce serverul apelează metoda `accept()`. Prin intermediul constructorului clasei `ServerSocket` se permite specificarea numărului de conexiuni care pot fi în coada de aşteptare; sistemul de operare va respinge orice cerere de conexiune ulterioară care apare în cazul în care coada este plină.

Constructorii clasei `ServerSocket` sunt:

Prototipul constructorului	Descriere
`ServerSocket``()`	Creează un socket pentru server, neataşat încă unui port.
`ServerSocket``(int port)`	Creează un socket pentru server, neataşat la portul indicat. Coada de aşteptare poate conţine cel mult 50 de cereri pentru acceptarea conexiunii.
`ServerSocket``(int port, int dimCoada)`	În plus faţă de constructorul precedent, se stabileşte şi dimensiunea cozii de aşteptare.
`ServerSocket``(int port, int dimCoada, InetAddress bindAddr)`	Al treilea parametru este obiect al clasei `InetAddress` şi indică adresa IP a maşinii pentru care se acceptă conexiuni. Dacă se transmite `null`, se vor accepta conexiuni pentru orice adresă a maşinii.

Toţi constructorii acestei clase pot arunca excepţia `IOException` şi, în afară de primul, şi excepţia `SecurityException`.

Cele mai importante metode publice ale clasei `ServerSocket` sunt cele din tabelul de mai jos:

Prototipul metodei	Descrierea metodei
`Socket `**`accept`**`() throws IOException`	Blochează execuţia firului de execuţie curent până când un client realizează o cerere pentru conexiune. Este returnat obiectul de tip `Socket` corespunzător conexiunii.
`void `**`close`**`() throws IOException`	Închide socketul. Orice metodă `accept()` blocată a unui fir de execuţie va arunca excepţia `SocketException`.
`InetAddress `**`getInetAddress`**`()`	Returnează adresa locală la care este ataşat socketul. Dacă nu se specifică nici o adresa locală în constructor, atunci se va returna `0.0.0.0`
`int `**`getLocalPort`**`()`	Returnează numărul portului la care este ataşat socketul sau `-1` dacă acesta nu este legat.

9.3.2. Clasa `Socket`

Un obiect `Socket` corespunde unei conexiuni de reţea TCP. Utilizând această clasă, un client poate stabili un canal de comunicaţie bazat pe fluxuri cu o maşină de la distanţă.

Structura generală a unui client care stabileşte o conexiune este foarte simplă. Acesta creează un obiect `Socket` care deschide o conexiune către un server şi apoi, prin intermediul aceluiaşi obiect, se realizează comunicarea cu serverul. Programul-client trebuie să cunoască adresa serverului, precum şi portul acestuia. Dacă la adresa şi portul indicat nu există nici un server activ, atunci stabilirea conexiunii va eşua şi va fi aruncată o excepţie `IOException`.

Clasa `Socket` este utilizată şi de server pentru comunicarea cu clienţii. Pentru fiecare client va exista un obiect de acest tip (obţinut în urma apelului metodei `accept()` din clasa `ServerSocket`), iar trimiterea, respectiv recepţionarea mesajelor pentru server sunt identice cu cele pentru client.

Crearea unui obiect `Socket` va realiza conexiunea cu serverul. Prezentăm cei mai semnificativi constructori:

- `Socket()` - creează un obiect `Socket` neconectat.
- `Socket(String gazda, int port) throws UnknownHostException, IOException` - creează un obiect `Socket` conectat la portul indicat al adresei gazdă specificate. `gazda` poate conţine numele serverului sau adresa IP. Portul trebuie să fie cuprins între valorile 1 şi 65535.
- `Socket(InetAddress adresa, int port) throws IOException` - faţă de constructorul precedent, se specifică adresa IP a serverului prin intermediul unui obiect de tip `InetAddress`.
- `Socket(String gazda, int port, InetAddress adresaLocala, int portLocal) throws IOException` - creează un obiect `Socket` legat la adresa locală (`adresaLocala`), portul local (`portLocal`) şi conectat la portul specificat (`port`) al adresei gazdă specificate (`gazda`). Dacă `adresaLocala` este `null`, atunci se utilizează adresa locală implicită. Dacă `portLocal` este 0, atunci se utilizează un port local arbitrar neutilizat.
- `Socket(InetAddress adresa, int port, InetAddress adresaLocala, int portLocal) throws IOException` - similar cu precedentul constructor, cu deosebirea că primul argument este un obiect din clasa `InetAddress`.

Metodele clasei `Socket` permit identificarea gazdei de la distanţă, a portului local şi a celui de la distanţă, precum şi extragerea de fluxuri în vederea comunicării bidirecţionale.

Pentru comunicarea într-o conexiune TCP, trebuie mai întâi stabilită conexiunea (crearea unui obiect `Socket`), apoi apelarea metodelor `getInputStream()` şi `getOutputStream()` pentru obţinerea fluxurilor de intrare, respectiv ieşire, cu ajutorul cărora se realizează comunicarea la distanţă. Aşadar, atât clientul, cât şi serverul vor avea câte un obiect `InputStream` şi `OutputStream`, în scopul comunicării.

În tabelul următor sunt date câteva metode uzuale ale clasei `Socket`:

Prototipul metodei	Descrierea metodei
`InputStream` **`getInputStream`**`() throws IOException`	Returnează un flux de intrare care permite citirea datelor din socket (trimise de celălalt partener).
`OutputStream` **`getOutputStream`**`() throws IOException`	Returnează un flux de ieşire care permite scrierea datelor în socket (către celălalt partener).
`void` **`close`**`() throws IOException`	Închide socketul curent, închizând conexiunea, şi eliberează orice resursă reţea şi sistem utilizată.
`InetAddress` **`getInetAddress`**`()`	Returnează adresa IP a maşinii de la distanţă.
`int` **`getPort`**`()`	Returnează numărul portului gazdei de la distanţă la care este conectat socketul curent.
`InetAddress` **`getLocalAddress`**`()`	Adresa maşinii locale de unde este conectat socketul curent.
`int` **`getLocalPort`**`()`	Returnează portul local, la care este conectat socketul curent.

9.3.3. O aplicaţie simplă client/server orientată conexiune

În această secţiune, vom prezenta un exemplu simplu de aplicaţie client/server orientată conexiune. Aplicaţia-client citeşte două numere reale de la tastatură, după care le trimite serverului. Acesta calculează maximul dintre cele două numere primite şi îl returnează.

Clasa `MaxServer` conţine definiţiile a trei metode statice:

- `public static void trimiteDateCatreClient(DataOutputStream out, String sir) throws IOException`
  Această metodă trimite şirul `sir` către fluxul de ieşire `out` utilizând metoda predefinită `writeUTF()`.
- `public static String primesteDateDeLaClient(DataInputStream in) throws IOException`
  Această metodă citeşte, utilizând metoda predefinită `readUTF()`, un `String` trimis prin fluxul de intrare `in`, pe care apoi îl returnează ca rezultat.
- `public static void main(String [] args)`
  Se încearcă crearea unui obiect `ServerSocket`, numit `server`, care să asculte la portul `2003`. În caz de eşec, se aruncă o excepţie `IOException`. În vederea creării socketului `s`, se apelează metoda `accept()` pentru obiectul `server`, apoi se creează cele două fluxuri `in` şi `out`, asociate socketului `s`. La pasul următor, se aşteaptă de la client cele două valori `double`, se calculează maximul acestora, care se trimite către client. În caz de eşec, se aruncă o excepţie `IOException`.

Fişierul `MaxServer.java` are următorul conţinut:

```
import java.net.*;
import java.io.*;
```

```
/** Clasa principala a aplicatiei */
public class MaxServer {
  /** Trimite date catre client. */
  public static void trimiteDateCatreClient(
      DataOutputStream out, String sir) throws IOException {
    out.writeUTF(sir); // trimite catre client sirul
    out.flush(); // goleste fluxul
    // afisarea unui mesaj la consola
    System.out.println("Am trimis catre client: " + sir);
  }

  /** Receptioneaza datele de la client. */
  public static String primesteDateDeLaClient(
      DataInputStream in) throws IOException {
    String sir = in.readUTF(); // obtine raspunsul de la client
    // afisarea unui mesaj la consola
    System.out.println("Am primit de la client: " + sir);
    return sir;
  }

  /** Metoda principala */
  public static void main(String [] args) {
      // declararea variabilelor
    DataInputStream in = null;
    DataOutputStream out = null;
    Socket s = null;
    ServerSocket server = null;
    try {
      // crearea socketului pentru server la portul 2003
      server = new ServerSocket(2003);
      // afisarea unui mesaj la consola
      System.out.println("Asteptam un client...");
      // acceptarea unei conexiuni cu un client
      s = server.accept();
      System.out.println(
        "S-a stabilit conexiunea cu clientul.");
      // obtinerea unui flux de intrare convenabil
      in = new DataInputStream
          (new BufferedInputStream (s.getInputStream()));
      // obtinerea unui flux de iesire convenabil
      out = new DataOutputStream
          (new BufferedOutputStream (s.getOutputStream()));
    } catch (IOException e) {
      // tratarea exceptiei
      System.err.println("Eroare la conectare: " + e);
    }
    // declararea altor variabile auxiliare
    String sirNumere = "";
    double nr1 = 0.0, nr2 = 0.0;
    double max = 0.0;
    try {
      // primirea unui numar
      sirNumere = primesteDateDeLaClient(in);
      Double tmp = Double.valueOf(sirNumere);
      nr1 = tmp.doubleValue();
      // primirea celuilalt numar
      sirNumere = primesteDateDeLaClient(in);
      tmp = Double.valueOf(sirNumere);
      nr2 = tmp.doubleValue();
      // calcularea maximului
      max = (nr1 < nr2)? nr2 : nr1;
      // trimiterea rezultatului
      trimiteDateCatreClient(out, Double.toString(max));
    } catch (IOException e) {
      // tratarea cazului de exceptie
      System.err.println(
        "Eroare la trimitere/primire date: " + e);
    }
  } // sfarsitul definitiei metodei main()
} // sfarsitul definitiei clasei MaxServer
```

Asemănător, clasa `MaxClient` conţine definiţiile a trei metode statice:

- `public static void trimiteDateCatreServer(DataOutputStream out, String sir) throws IOException`
  Această metodă trimite şirul `sir` către fluxul de ieşire `out`, utilizând metoda predefinită `writeUTF()`.
- `public static String primesteDateDeLaServer(DataInputStream in) throws IOException`
  Această metodă citeşte, utilizând metoda predefinită `readUTF()`, un `String` trimis prin fluxul de intrare `in`, pe care apoi îl returnează ca rezultat.
- `public static void main(String [] args)`
  Se încearcă crearea unui obiect de tip `Socket`, numit `s`, asociat maşinii locale (de adresă `127.0.0.1`) la portul `2003`. În caz de eşec, se aruncă o excepţie `IOException`. Apoi se creează cele două fluxuri `in` şi `out`, asociate socketului `s`. Apoi, se aşteaptă de la tastatură introducerea a două valori `double` care se trimit către server apelând metoda `trimiteDateCatreServer()`. În variabila `rezultat` se va primi rezultatul returnat de apelul metodei `primesteDateDeLaServer()`, care va fi afişat la consolă. În caz de eşec, se aruncă o excepţie `IOException`.

Conţinutul fişierului `MaxClient.java` este următorul:

```
import java.net.*;
import java.io.*;

/** Aplicatia client MaxClient */
public class MaxClient {
  /** Trimite datele catre server. */
  public static void trimiteDateCatreServer(
     DataOutputStream out, String sir) throws IOException {
    out.writeUTF(sir); // trimite catre server sirul
    out.flush(); // goleste sirul din flux
    // afisarea unui mesaj la consola
    System.out.println("Am trimis catre server: " + sir);
  }

  /** Primeste datele de la server. */
 public static String primesteDateDeLaServer(DataInputStream in)
     throws IOException  {
   String sir = in.readUTF(); // obtine raspunsul de la server
    System.out.println("Am primit de la server: " + sir);
    return sir;
  }

  /** Metoda principala a aplicatiei */
  public static void main(String [] args) {
    // declararea datelor locale
    DataInputStream in = null;
    DataOutputStream out = null;
    Socket s = null;
    try {
      /* stabilirea unei conexiuni cu
      serverul local de la portul 2003 */
      s = new Socket("127.0.0.1", 2003);
      // afisarea unui mesaj de succes
      System.out.println("Ne-am conectat la server.");
      // obtinerea unui flux de intrare convenabil
      in = new DataInputStream
          (new BufferedInputStream(s.getInputStream()));
      // obtinerea unui flux de iesire convenabil
      out = new DataOutputStream(
          new BufferedOutputStream(s.getOutputStream()));
    } catch (IOException e) {
      // tratarea exceptiei
      System.err.println("Eroare la conectare: " + e);
      System.exit(1);
    }
```

```
      // declararea altor date auxiliare
      double nr1 = 0.0, nr2 = 0.0;
      BufferedReader tastatura;
      String linie;
      try {
        /* obtinerea unui flux de la intrarea standard
            (de la tastatura) */
        tastatura = new BufferedReader(new InputStreamReader(
          System.in));
        System.out.flush();
        // citirea primului numar de la tastatura
        System.out.print("Dati primul numar: ");
        linie = tastatura.readLine();
        Double tmp = Double.valueOf(linie);
        nr1 = tmp.doubleValue();
        System.out.flush();
        // citirea celui de-al doilea numar
        System.out.print("Dati al doilea numar: ");
        linie = tastatura.readLine();
        tmp = Double.valueOf(linie);
        nr2 = tmp.doubleValue();
        tastatura.close(); // inchiderea intrarii standard
      } catch (IOException e) {
        // tratarea cazului de exceptie
       System.err.println("Citire gresita de la tastatura: " + e);
      }
      // comunicarea cu serverul
      String rezultat = "";
      try {
        // trimiterea datelor
        trimiteDateCatreServer(out, Double.toString(nr1));
        trimiteDateCatreServer(out, Double.toString(nr2));
        // primirea rezultatului
        rezultat = primesteDateDeLaServer(in);
      } catch (IOException e) {
        System.err.println(
          "Eroare la trimitere/primire date: " + e);
      }
      // afisarea rezultatului
      System.out.println("Rezultat: " + rezultat);
    } // sfarsitul definitiei metodei main()
  } // sfarsitul definitiei clasei MaxClient
```

Cele două programe Java sunt compilate şi executate cu ajutorul comenzilor `javac`, respectiv `java`. Mai întâi se porneşte în execuţie serverul, apoi clientul. Pentru client va trebui să introducem de la tastatură două numere.

O posibilă execuţie a clientului poate fi:

```
Ne-am conectat la server.
Dati primul numar: 123.230
Dati al doilea numar: 0.07
Am trimis catre server: 123.23
Am trimis catre server: 0.07
Am primit de la server: 123.23
Rezultat: 123.23
```

iar a serverului:

```
Asteptam un client...
S-a stabilit conexiunea cu clientul.
Am primit de la client: 123.23
Am primit de la client: 0.07
Am trimis catre client: 123.23
```

În acest exemplu, serverul acceptă o conexiune, deserveşte cererea, apoi îşi încheie rularea. În practică, la apariţia unui client se creează un nou fir de execuţie care deserveşte cererea, iar firul principal se ocupă doar de acceptarea conexiunilor. De regulă, un server se execută la infinit, eventual acesta poate primi anumite comenzi de la tastatură pentru oprire, reiniţializare sau repornire.

9.4. Programarea reţelelor prin intermediul datagramelor

În Java, socketurile pentru protocoale fără conexiune (în speţă, UDP) sunt implementate de clasele `DatagramPacket` şi `DatagramSocket`.

9.4.1. Clasa `DatagramPacket`

Clasa `DatagramPacket` corespunde unui pachet de date numit şi *datagramă*. Aceasta permite specificarea adresei destinatarului şi a informaţiei transportate. Nu este asigurată primirea mesajului de către destinatar, însă rata de pierdere este extrem de mică. De asemenea, nu este asigurată nici recepţionarea mesajelor în ordinea în care acestea au fost emise. De exemplu, destinatarul poate primi în această ordine: datagrama a doua, datagrama a treia şi apoi pe prima. Însă comunicarea datagramelor este mai rapidă faţă de cea orientată conexiune, tocmai datorită faptului că nu se realizează verificări suplimentare (confirmarea primirii de către destinatar) şi a refacerii ordinii.

O aplicaţie care recepţionează o datagramă poate extrage din respectivul pachet atât informaţia trimisă, cât şi adresa expeditorului (adresa maşinii şi numărul portului).

Există două tipuri de constructori pentru clasa `DatagramPacket`, după scopul de întrebuinţare a pachetelor: pentru transmitere sau pentru recepţionare. Constructorii pentru recepţionarea datagramelor sunt:

- `DatagramPacket(byte[] buf, int lungime)`
  Parametrul `buf` desemnează zona de memorie în care se va copia conţinutul mesajului (aceasta trebuie să fie alocată), iar `lungime`, numărul maxim de octeţi pe care îl poate avea respectivul mesaj. Trebuie să avem grijă ca valoarea dată de `lungime` să nu depăşească capacitatea lui `buf` (`buf.length`).
- `DatagramPacket(byte[] buf, int deplasament, int lungime)`
  Ca şi mai sus, parametrul `buf` este utilizat pentru a indica zona de memorie în care se va salva mesajul primit. Acesta va fi stocat în respectivul buffer, începând cu poziţia `deplasament` şi va conţine cel mult `lungime` octeţi. De asemenea, trebuie să avem grijă să nu se depăşească capacitatea bufferului.

În plus, pentru transmiterea datagramelor trebuie să mai indicăm adresa destinatarului. Prin urmare, constructorii pentru trimiterea datagramelor vor avea, în plus faţă de cei pentru recepţionare doi parametri: unul pentru adresa maşinii şi unul pentru indicarea portului serverului. Cei mai importanţi constructori ai clasei `DatagramPacket` utilizaţi pentru transmiterea datagramelor sunt:

- `DatagramPacket(byte[] buf, int lungime, InetAddress adresa, int port)`
- `DatagramPacket(byte[] buf, int deplasament, int lungime, InetAddress adresa, int port)`

Parametrii `buf`, `lungime` şi `deplasament` au aceeaşi semnificaţie ca la constructorii pentru primirea datagramelor, cu deosebirea că se referă la transmiterea mesajului (lungime va indica numărul de caractere pe care îl va avea mesajul). În plus, parametrul `adresa` indică adresa maşinii serverului, iar `port` specifică portul la care aşteaptă datagrame respectivul server.

Pachetul va fi trimis către portul şi gazda specificate. Trebuie să existe un server UDP care ascultă la portul specificat pentru trimiterea pachetelor. Atenţie! Numerele de port UDP şi TCP sunt complet independente, deci putem avea şi un server UDP, şi un server TCP care ascultă la acelaşi port.

În funcţie de tipul constructorului apelat se va trimite, respectiv recepţiona o datagramă. Dacă pachetul este primit, atunci adresa corespunde gazdei, iar dacă pachetul a fost creat pentru transmisie, atunci adresa corespunde maşinii-destinaţie. Conţinutul şi adresa unui obiect `DatagramPacket` pot fi interogate şi modificate cu metodele:

Prototipul metodei	Descrierea metodei
`InetAddress` **`getAddress`**`()`	Returnează adresa IP a datagramei.
`int` **`getPort`**`()`	Returnează portul conţinut de datagramă.
`byte[]` **`getData`**`()`	Returnează conţinutul datagramei.
`int` **`getLength`**`()`	Întoarce dimensiunea mesajului datagramei (numărul de octeţi).
`void` **`setAddress`**`(InetAddress adresa)`	Stabileşte adresa pentru datagrama curentă.
`void` **`setData`**`(byte[] buf)`	Modifică conţinutul mesajului.
`void` **`setLength`**`(int lungime)`	Modifică dimensiunea mesajului.

9.4.2. Clasa `DatagramSocket`

Clasa `DatagramSocket` corespunde unui socket pentru datagrame, prin intermediul căruia se vor putea transmite, respectiv recepţiona astfel de pachete. Această clasă se poate utiliza atât pentru trimiteri, cât şi pentru primiri de datagrame (obiecte `DatagramPacket`). Ca şi în cazul unui server TCP, unul UDP trebuie să asculte la un număr de port particular, între 1 şi 65535 (porturile cuprinse între 1 şi 1024 sunt rezervate, de obicei, pentru aplicaţii sistem). Deoarece UDP nu este orientat conexiune, se va crea un singur socket (obiect `DatagramSocket`) pentru trimiterea pachetelor către diferite destinaţii şi primirea pachetelor de la diferite surse.

Un obiect `DatagramSocket` va recepţiona la un anumit port UDP de pe maşina locală pachetele care sosesc. Numărul portului se poate specifica sau, eventual, sistemul de operare poate să-l asigneze. De obicei, serverul va alege un port particular cu care să opereze, iar clienţii vor permite stabilirea unui port aleator. Numărul de port şi adresa expeditorului vor fi automat inserate în fiecare pachet trimis. Există şi posibilitatea legării unui obiect `DatagramSocket` la o adresă locală particulară, pe o maşină căreia îi sunt asignate mai multe adrese IP.

Cei mai utilizaţi constructori ai clasei `DatagramSocket` sunt:

- `DatagramSocket() throws SocketException`

  Se creează un obiect `DatagramSocket` cu un număr de port ales aleator.
- `DatagramSocket(int port) throws SocketException`

  Acest constructor creează un socket UDP care ascultă la numărul de port specificat.
- `DatagramSocket(int port, InetAddress locala) throws SocketException`

  Se creează un obiect `DatagramSocket` care ascultă la numărul de port specificat şi este legat la adresa locală indicată. Pentru maşinile cărora le sunt asignate mai multe adrese IP, pachetele sunt trimise utilizând adresa IP specificată.

Clasa `DatagramSocket` furnizează metode pentru trimiterea şi primirea de obiecte `DatagramPacket`, închiderea socketului, obţinerea adresei locale, setarea timpului de aşteptare pentru primire etc.:

Prototipul metodei	Descrierea metodei
`void` **`send`**`(DatagramPacket pachet) throws IOException`	Trimite datagrama `pachet` la destinatar. În cazul în care nu se cunoaşte destinatarul, va apărea o excepţie `IOException`.
`void` **`receive`** `(DatagramPacket pachet) throws IOException`	Recepţionează o singură datagramă în obiectul `pachet`. Ulterior se poate cerceta cine este expeditorul, care este lungimea mesajului etc. Execuţia *thread*-ului este blocată până când se recepţionează un pachet sau se scurge timpul de aşteptare.
`InetAddress` **`getLocalAddress`**`()`	Returnează adresa locală la care este ataşat socketul.

➔

`int` **`getLocalPort`**`()`	Întoarce portul la care aşteaptă socketul curent.
`void` **`close`**`()`	Închide socketul, eliberând astfel resursele ocupate.
`void` **`setSoTimeout`**`(int timpDeAsteptare) throws SocketException`	Setează timpul de aşteptare în milisecunde a unui socket la valoare specificată. Metoda `receive()` se va bloca doar pentru timpul de aşteptare specificat pentru primirea unui pachet UDP, după care va arunca o excepţie `InterruptedException`. Setarea timpului la 0 va dezactiva această facilitate.
`int` **`getSoTimeout`**`() throws SocketException`	Returnează timpul de aşteptare a socketului curent sau 0, dacă acesta nu este setat.
`void` **`setSendBufferSize`**`(int lungime) throws SocketException`	Setează lungimea bufferului socketului pentru trimitere la valoarea specificată. Pachetele care au un număr mai mare de octeţi decât această valoare nu vor putea fi trimise.
`int` **`getSendBufferSize`**`() throws SocketException`	Returnează lungimea curentă a bufferului socketului pentru trimitere.
`void` **`setReceiveBufferSize`**`(int lungime) throws SocketException`	Setează lungimea bufferului socketului pentru recepţionare la valoarea specificată. Pachetele mai mari de această valoare nu vor putea fi primite.
`int` **`getReceiveBufferSize`**`() throws SocketException`	Returnează lungimea curentă a bufferului socketului pentru recepţionare.
`void` **`connect`**`(InetAddress adresa, int port) throws SocketException`	Pseudoconectează acest socket la adresa şi portul specificat la distanţă. Schimbul de pachete se va realiza doar cu socketul de la adresa şi portul respectiv. În general, această metodă se utilizează din motive de performanţă şi nu este cerută pentru operaţiile uzuale.
`void` **`disconnect`**`()`	Întrerupe pseudoconexiunea realizată anterior
`InetAddress` **`getInetAddress`**`()`	Returnează obiectul `InetAddress` la care este pseudoconectat socketul sau `null`, dacă acesta nu este conectat.
`int` **`getPort`**`()`	Întoarce numărul portului la care este pseudoconectat socketul sau -1, dacă acesta nu este conectat.

La crearea unui obiect `DatagramSocket` sau la setarea acestuia se poate arunca o excepţie `SocketException`, iar dacă apar probleme la trimiterea sau primirea pachetelor, se va arunca excepţia `IOException`.

Managerul de securitate (*SecurityManager*) poate restricţiona accesul transmisiunilor şi primirilor datagramelor. De exemplu, appleturile nesigure nu pot trimite şi nici primi date de la alt server decât serverul iniţial. Restricţiile de primire sunt rezolvate prin verificarea adresei-sursă a fiecărui pachet primit. Dacă un pachet vine dintr-o sursă nevalidă, atunci acesta este eliminat şi metoda `receive()` continuă să aştepte pachete valide.

9.4.3. O aplicaţie simplă client/server neorientată conexiune

Comunicarea cu un protocol fără conexiune este mai simplă decât cu unul orientat conexiune, deoarece atât clientul, cât şi serverul recurg la obiecte de tip `DatagramSocket`. De regulă, un server care lucrează cu datagrame trebuie să parcurgă următoarele etape:

- crearea unui obiect de tip `DatagramSocket`, asociat cu un număr de port specificat;
- crearea unui obiect `DatagramPacket`;
- cu ajutorul obiectului de tip `DatagramSocket`, se stochează o datagramă în obiectul de tip `DatagramPacket`;
- se procesează cererea;
- se completează o nouă datagramă;
- se trimite datagrama completată la pasul anterior.

La partea client, ordinea este oarecum inversată: mai întâi se trimite o datagramă, după care se aşteaptă recepţionarea răspunsului.

Vom aplica şablonul de mai sus în vederea scrierii unei aplicaţii client/server care utilizează datagramele. Clientul va cere serverului timpul curent şi serverul îl va furniza.

Pentru server am creat clasa `TimpServer`. Metoda `main()` începe prin crearea unui obiect `DatagramSocket` care utilizează portul `2003`. Apoi creăm un obiect `DatagramPacket` care va conţine datele primite prin intermediul obiectul de tip `DatagramSocket`. Serverul posedă o buclă infinită în care primeşte cereri de la clienţi, utilizând metoda `receive()` pentru obiectul `DatagramSocket`, şi apoi trimite răspunsuri.

Codul-sursă al clasei `TimpServer` este:

```
import java.net.*;
import java.io.*;

public class TimpServer {
  static DatagramSocket socket;

  public static void main(String[] argv) {
    try {
      // crearea unui socket UDP la portul 2003
      socket = new DatagramSocket(2003);
    } catch (SocketException e) {
       System.err.println("Nu putem crea socketul" +
          e.getMessage());
       System.exit(1);
    }
    // pregatirea pentru receptionarea datagramelor
```

```
      DatagramPacket datagrama;
      datagrama = new DatagramPacket(new byte[1], 1);
      System.out.println("Asteptam clienti...");
      // intrarea in bucla infinita
      while (true) {
         try {
            // receptionarea unei datagrame
            socket.receive(datagrama);
            // raspunde la cererea primita
            raspunde(datagrama);
         } catch (IOException e) {
            System.err.println(
               "Nu putem primi sau trimite datagrama"
               + e.getMessage());
         }
      }
   } // sfarsitul definitiei metodei main()


   static void raspunde(DatagramPacket cerere) {
      // obtinerea unui flux de iesire convenabil
      ByteArrayOutputStream baos;
      baos = new ByteArrayOutputStream();
      DataOutputStream out = new DataOutputStream(baos);
      // scrierea in flux a timpului curent in milisecunde
      try {
         out.writeLong(System.currentTimeMillis());
      } catch (IOException e) {
         System.err.println("Nu putem trimite timpul curent"
            + e.getMessage());
      }

      // construirea raspunsului
      DatagramPacket raspuns;
      // stabilirea mesajului
      byte[] data = baos.toByteArray();
      // crearea datagramei
      raspuns = new DatagramPacket(data, data.length,
         cerere.getAddress(), cerere.getPort());
      // trimiterea efectiva a pachetului
      try {
        socket.send(raspuns);
      }
      catch (IOException e) {
        System.err.println("Nu putem trimite datagrama"
            + e.getMessage());
      }
    } // sfarsitul definitiei metodei raspunde()
  } // sfarsitul definitiei clasei TimpServer
```

Metoda `raspunde()` se ocupă cu trimiterea răspunsurilor. Vom scrie **timpul** curent ca o valoare `long` într-un şir de octeţi. Apoi, pregătim trimiterea şirului de octeţi prin crearea unui obiect `DatagramPacket` ce încapsulează şirul, adresa şi numărul de port al clientului care solicită timpul curent. În final, se trimit datele apelând metoda `send()` a obiectului `DatagramSocket`.

Codul-sursă corespunzător aplicaţiei-client este:

```
import java.net.*;
import java.io.*;
import java.util.*;

public class TimpClient {
  public static void main(String argv[]) {
    if (argv.length != 1) {
      System.out.println(
        "Dati comanda: ClientTimp numeServer");
      System.exit(1);
    }
    DatagramSocket socket = null;
    try {
      socket = new DatagramSocket();
    } catch (SocketException e) {
      System.err.println("Nu putem crea socketul" +
         e.getMessage());
      System.exit(1);
    }
    long timp = 0;
    try {
      byte[] buffer = new byte[1];
      socket.send(new DatagramPacket(buffer, 1,
         InetAddress.getByName(argv[0]), 2003));
      DatagramPacket raspuns = new DatagramPacket(new byte[8], 8);
      socket.receive(raspuns);
      ByteArrayInputStream baos;
      baos = new ByteArrayInputStream(raspuns.getData());
      DataInputStream dis = new DataInputStream(baos);
      timp = dis.readLong();
    } catch (IOException e) {
      System.err.println(
        "Nu putem trimite sau primi datagrame! "
         + e.getMessage());
      }
      System.out.println("Timpul primit de la server este "
         + new Date(timp));
      socket.close();
   } // sfarsitul definitiei metodei main()
} // sfarsitul definitiei clasei ClientTimp
```

Pentru client am implementat clasa `ClientTimp`. Acesta verifică mai întâi numărul corect de argumente (trebuie să fie un argument cu adresa IP). Apoi se creează un obiect `DatagramSocket` pentru comunicarea cu serverul. Urmează crearea unui obiect `DatagramPacket` care va conţine cererea ce trebuie trimisă către server utilizând metoda `send()`. Apoi, creăm încă un obiect `DatagramPacket` pentru primirea răspunsului de la server cu ajutorul apelului `receive()`. În final, clientul afişează timpul curent şi închide socketul (indiferent dacă se aruncă o excepţie sau nu).

9.4.4. Clasa **InetAddress**

Datele pot fi trimise prin reţea, utilizând adresa IP specifică maşinii-destinaţie. Clasa `InetAddress` furnizează acces abstract la adresele IP. Avantajul utilizării acestei clase pentru reprezentarea unei adrese IP într-un întreg pe 32 biţi este că aplicaţiile vor fi transparent portabile la IPv6 (IP versiunea 6), care furnizează adrese pe 128 de biţi.

Nu există constructori pentru această clasă. Instanţele sunt create prin intermediul metodelor statice `getByName()`, `getLocalHost()` şi `getAllByName()`. Acestea au sintaxa:

- `InetAddress getLocalHost() throws UnknownHostException` – returnează un obiect de tip `InetAddress` corespunzător maşinii locale.
- `InetAddress getByAddress(byte[] addr)` – întoarce un obiect de tip `InetAddress` corespunzător adresei de IP date în tabloul specificat. Pentru adresele IPv4 se va transmite un tablou de 4 elemente (octeţi), iar pentru IPv6, de 16 elemente.
- `InetAddress getByName(String host) throws UnknownHostException` – pentru maşina-gazdă specificată se construieşte un obiect de tip `InetAddress`. *Host*-ul poate fi specificat prin nume (de exemplu, `dil.info.uaic.ro`) sau prin adresa IP (de exemplu, `193.231.30.182`).
- `InetAddress [] getAllByName(String host) throws UnknownHostException` – returnează un vector de obiecte `InetAddress` corespunzător fiecărei adrese IP cunoscute pentru *host*-ul specificat. De obicei, siturile Web cu trafic mare au mai multe adrese IP pentru un singur nume de *host*.

Există două excepţii care pot fi aruncate de metodele statice de mai sus:

- `UnknownHostException`, care este subclasă a clasei `IOException` şi indică faptul că *host*-ul căutat nu a fost identificat cu succes;

- `SecurityException`, care poate fi aruncată dacă managerul de securitate nu permite o operaţie specifică. Un applet (nesigur) poate doar să construiască un obiect `InetAddress` pentru numele gazdei corespunzătoare serverului Web de unde provine codul său.

Clasa `InetAddress` are următoarele metode instanţă (adică asociate unui obiect al clasei, şi nu clasei):

Prototipul metodei	Descrierea metodei
`byte [] getAddress()`	Întoarce un vector de octeţi corespunzător adresei IP. Octeţii respectă ordinea reţelei, adică octetul semnificativ este primul.
`String getHostName()`	Returnează numele maşinii-gazdă. În cazul în care nu se poate afla numele, întoarce adresa IP.
`String getHostAddress()`	Întoarce adresa IP.
`String toString()`	Converteşte adresa IP într-un şir de caractere.

Exemplul 9.4.1. Următorul program Java afişează adresa IP locală.

```
import java.net.*;

public class AdresaIPLocala {
  public static void main(String args[]) {

    InetAddress adresaIPLocala = null;
    try {
     adresaIPLocala = InetAddress.getLocalHost();
     System.out.println("adresaIPLocala = " + adresaIPLocala);
    } catch (UnknownHostException e) {
      System.err.println("Nu putem gasi gazda: " +
      e.toString());
    }
  }
}
```

O posibilă execuţie a acestui program poate fi:

```
adresaIPLocala = felix/192.168.10.3
```

unde `felix` este numele calculatorului, iar `192.168.10.3` este adresa IP a maşinii locale.

Exemplul 9.4.2. Programul Java afişează adresa IP a unei maşini la distanţă, plecând de la numele de domeniu.

```
import java.net.*;
import java.io.*;
```

```
public class adresaIPLaDistanta {
  public static void main(String args[]) {
    BufferedReader tastatura;
    tastatura = new BufferedReader(new InputStreamReader(
      System.in),1);
    System.out.print("Dati un nume de domeniu: ");
    String linie = "";
    // citim numele de domeniu reprezentat ca un String
    try {
      System.out.flush();
      linie = tastatura.readLine();
      tastatura.close();
    } catch (IOException e)  {
      System.out.println("Exceptie la citirea de la tastatura"
         + e);
      System.exit(2);
    }
    InetAddress adresaIP = null;
    try {
      adresaIP = InetAddress.getByName(linie);
      System.out.println(adresaIP);
    } catch (UnknownHostException e) {
      System.err.println("Nu putem gasi gazda: " + e);
    }
  }
}
```

O posibilă execuţie a acestui program este:

```
Dati un nume de domeniu: webgroup.info.uaic.ro
webgroup/193.231.30.227
```

9.4.5. Clasa `URL`

Clasa `URL` furnizează acces de nivel înalt la datele de la distanţă. Un obiect de tip `URL` încapsulează o specificare a unui locator uniform de resurse (URL). Odată creat un astfel de obiect, acesta poate fi utilizat pentru accesarea datelor locaţiei indicate. Un URL permite accesul la date fără a fi nevoie de detaliile protocolului folosit, cum ar fi HTTP sau FTP. Pentru anumite tipuri de date, un obiect de tip URL furnizează un mod de a obţine date deja încapsulate în alte obiecte asemănătoare. De exemplu, un URL poate furniza date JPEG încapsulate într-un obiect `ImageProducer` sau date text încapsulate într-un obiect `String`.

Cel mai utilizat constructor pentru clasa `URL` este cel cu un parametru de tip `String` care conţine adresa URL corespunzătoare. În cazul în care parametrul nu conţine un URL valid, se va arunca excepţia `MalformedURLException`.

De exemplu, instanţierea clasei `URL` se poate realiza astfel:

```
try {
  URL sitFacultate = new URL(
    "http://www.infoiasi.ro/");
} catch (MalformedURLException e) {
  System.err.println("URL gresit! " + e.getMessage());
}
```

Odată creat un obiect de tip `URL`, se pot utiliza metodele sale pentru obţinerea informaţiilor încapsulate:

Prototipul metodei	Descrierea metodei
`String` **`getFile`**`()`	Returnează numele fişierului.
`String` **`getHost`**`()`	Extrage numele maşinii-gazdă.
`String` **`getPort`**`()`	Întoarce portul utilizat.
`String` **`getProtocol`**`()`	Returnează numele protocolului.
`String` **`getRef`**`()`	Întoarce referinţa (sau ancora).
`String` **`getQuery`**`()`	Extrage şirul de interogare.
`boolean` **`sameFile`**`(URL altul)`	Testează dacă obiectul curent şi cel specificat indică aceeaşi resursă (fişier).
`URLConnection` **`openConnection`**`()` `throws IOException`	Realizează o conexiune la resursa indicată.
`final InputStream` **`openStream`**`()` `throws IOException`	Deschide o conexiune la adresa referită şi întoarce obiectul `InputStream` corespunzător.
`final Object` **`getContent`**`()`	Returnează conţinutul resursei referite sub forma unui obiect, al cărui tip este convenabil ales în funcţie de tipul fişierului.

Exemplul 9.4.3. Appletul de mai jos citeşte o adresă URL şi întoarce numele protocolului, numele maşinii-gazdă, portul, numele fişierului şi referinţa (ancora).

```
import java.applet.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;

public class InfoURL extends Applet implements ActionListener {
  // datele membre
  URL url;
  String sirURL;
  Button afiseaza;
  TextField numeCamp;
```

```
   /** initializeaza appletul */
   public void init() {
     // crearea unui buton
     afiseaza = new Button("Dati adresa URL");
     add(afiseaza);
     afiseaza.addActionListener(this);
     // crearea unui camp de text
     numeCamp = new TextField(60);
     add(numeCamp);
   }

   public void actionPerformed(ActionEvent event) {
     // preluam URL introdus in campul text
     sirURL = numeCamp.getText();
     try {
       // instantiem un obiect de tip URL
       url = new URL(sirURL);
       // obtinem numele fisierului
       String numeFisier = url.getFile();
       System.out.println("Numele fisierului este " + numeFisier);
       // obtinem numele masinii-gazda
       String numeHost = url.getHost();
       System.out.println("Numele hostului este " + numeHost);
       // obtinem numele portului
       int numarPort = url.getPort();
       System.out.println("Numarul portului este " + numarPort);
       // obtinem numele protocolului
       String numeProtocol = url.getProtocol();
       System.out.println("Numele protocolului este " +
         numeProtocol);
       // extragem referinta (ancora)
       String numeReferinta = url.getRef();
       System.out.println("Numele referintei este " +
         numeReferinta);
     } catch (MalformedURLException e) {
       // tratarea exceptiei
       System.err.println("Eroare:" + e.getMessage());
     }
   }
}
```

Dacă se va introduce în câmpul text al appletului adresa: `http://www.infoiasi.ro:8080/fcs/index.html#admitere`, atunci se va afişa la consolă:

```
Numele fisierului este /fcs/index.html
Numele hostului este www.infoiasi.ro
Numarul portului este 8080
Numele protocolului este http
Numele referintei este admitere
```

Continuăm cu citirea unui fişier de pe un server aflat la distanţă care se realizează în mod similar cu citirea unui fişier local (datorită puterii bibliotecilor Java). În loc de numele fişierului vom specifica adresa URL a acestuia.

Pentru a accesa conţinutul unui fişier linie cu linie în loc să-l obţinem încapsulat într-un obiect, putem utiliza metoda `openStream()` a clasei `URL` sau metoda `getInputStream()` din clasa `URLConnection`, care întoarce o referinţă la un obiect din clasa `InputStream`.

Pentru crearea unui obiect din clasa `URLConnection`, se utilizează metoda `openConnection()` din clasa URL. Dacă nu este deja deschisă o conexiune, metoda `openConnection()` deschide o conexiune pentru acest URL prin apelarea metodei `openConnection()` din clasa `URLStreamHandler`. Un obiect `URLStreamHandler` pentru protocolul URL este creat de constructorul `URL`. În pachetul `java.net`, subclasele clasei `URLStreamHandler` se ocupă cu diferite protocoale pentru comunicaţie.

Exemplul 9.4.4. Programul Java de mai jos citeşte de la tastatură o adresă URL şi afişează conţinutul fişierului pe ecran. Aplicaţia permite doar afişarea documentelor de tip text.

```
import java.io.*;
import java.net.*;

public class ContinutTextURL {
  // declararea datelor membre
  private BufferedReader tastatura, fluxIntrare;

  public static void main(String[] args) {
    ContinutTextURL continut = new ContinutTextURL();
    continut.afisare();
  }

  /** Afiseaza continut de la adresa data de URL */
  private void afisare() {
    String sirURL = "";
    String linie = "";

    // obtinerea unui flux de la tastatura
    tastatura = new BufferedReader(new InputStreamReader
      (System.in), 1);
    try {
      // obtinerea adresei URL de la tastatura
      sirURL = prompt("Dati o adresa URL (ex.: " +
         "http://www.infoiasi.ro/):");
      // obtinerea unei instante a clasei URL
      URL adresaURL = new URL(sirURL);
      // crearea unei conexiuni
      URLConnection conexiune = adresaURL.openConnection();
      // obtinerea unui flux pentru citirea continutului resursei
      fluxIntrare = new BufferedReader(new
         InputStreamReader(conexiune.getInputStream()));
      // cat timp putem citi cate o linie, o afisam la consola
      while ((linie = fluxIntrare.readLine()) != null) {
        System.out.println(linie);
      }
    } // tratarea exceptiilor
     catch (MalformedURLException e) {
      System.err.println("URL gresit: " + sirURL + "\n" + e);
      System.exit(2);
    } catch (IOException e) {
      System.err.println("Eroare la stabilirea conexiunii: " + e);
      System.exit(1);
    }
  }

  /** Citeste o linie de la tastatura */
  private String prompt(String mesaj) {
    String raspuns = "";
    try {
      // afisarea unui mesaj
      System.out.print(mesaj);
      // golirea fluxului de iesire
      System.out.flush();
      // citirea unei linii de la tastatura
      raspuns = tastatura.readLine();
    } catch (IOException e) {
      System.err.println("Eroare la citirea de la tastatura:"
        + e);
      System.exit(2);
    }
    // intoarcerea rezultatului
    return raspuns;
  }
}
```

Dacă indicăm adresa `http://localhost/salut.html`, vom obţine conţinutul fişierului `salut.html` de pe serverul Web local, care poate avea următorul conţinut:

```
<html>
<head>
  <title>Salutari</title>
</head>
<body>
Salutari de pe serverul Web.
</body>
</html>
```

Cel mai înalt nivel de funcţionalitate disponibil la un obiect de tip `URL` este furnizat de metoda `getContent()`. Aceasta încearcă să determine tipul de date din fişierul specificat de URL şi apoi returnează conţinutul încapsulat într-un obiect convenabil pentru acel tip de date. De exemplu, dacă fişierul conţine date în format GIF (cu extensia `.gif`), metoda `getContent()` returnează un obiect din clasa `ImageProducer`. Dacă tipul de date nu este explicit specificat, metoda va încerca să determine tipul după extensia fişierului şi posibil, de asemenea, din conţinutul acestuia (în general, din primele linii ale sale).

Numele tipurilor de date pe care Java le utilizează sunt conforme cu bine cunoscuta schemă pentru tipuri de date: MIME. Acestea sunt de forma `tip/subtip`.

În tabelul de mai jos se găsesc câteva extensii de fişier recunoscute de Java şi tipul MIME al acestora:

Extensie fişier	Tipul MIME
`.au`	`audio/basic`
`.c`	`text/plain`
`.cpp`	`text/plain`
`.dvi`	`application/x-dvi`
`.exe`	`application/octet-stream`
`.gif`	`image/gif`
`.html`	`text/html`
`.java`	`text/plain`
`.txt`	`text/plain`
`.zip`	`application/zip`

Exemplul 9.4.5. Vizualizarea textului şi a imaginilor din fişiere:

```
import java.applet.*;
import java.net.*;
import java.awt.*;
import java.awt.image.*;
import java.io.*;

public class VizualizareTextSiImagine extends Applet {
  Image img;
  String str;
```

```
    public void init() {
      try  {
        // crearea unui URL pentru o imagine
        URL fisierImagine = new URL(getDocumentBase(), "t1.gif");
        // obtinerea continutului grafic
        img = this.createImage((ImageProducer) fisierImagine.
          getContent());
        // crearea unui URL pentru un fisier text
        URL fisierText = new URL(getDocumentBase(), "a.txt");
        // obtinerea continutului
        str = fisierText.getContent().toString();
      } // tratarea exceptiilor
      catch (MalformedURLException e) {
        System.err.println("Eroare:" + e.getMessage());
      } catch (IOException e) {
        System.err.println("Eroare:" + e.getMessage());
      }
      // actualizarea appletului (redesenare)
      repaint();
    }

    public void paint(Graphics g) {
      // desenarea imaginii
      g.drawImage(img, 0, 0, this);
      // desenarea continutului text al fisierului
      g.drawString(str, 10, 200);
    }
  }
```

Dacă numele fişierului nu se termină cu o extensie cunoscută, sunt examinaţi primii octeţi. Dacă aceştia „se potrivesc" cu semnătura unui tip cunoscut, atunci se presupune că respectivul document este de acel tip.

Câteva combinaţii de octeţi care sunt recunoscute sunt:

- `GIF8` pentru `image/gif`;
- `#def` pentru `image/xbitmap`;
- `! XPM2` pentru `image/xpixmap`.

9.4.6. Obţinerea unei pagini Web prin intermediul unui socket

Pentru a stabili o conexiune HTTP trebuie să creăm un socket pentru comunicare, să-l conectăm la serverul Web (trebuie să specificăm numele serverului sau adresa IP şi portul acestuia, implicit fiind 80) şi să-i trimitem mesaje conforme cu specificaţiile protocolului HTTP. Astfel, serverul va putea descifra cererile emise, iar ca rezultat vom obţine mesaje HTTP. Acestea sunt alcătuite din antet (engl. *header*) şi conţinutul mesajului. Cele două părţi sunt despărţite de o linie vidă.

Exemplul 9.4.6. Aplicaţia va aştepta citirea URL-urilor, iar după ce am specificat unul de la tastatură, va fi afişat, în cazul în care acesta este valid, conţinutul paginii Web.

```
import java.net.*;
import java.io.*;

public class ObtinePaginaWeb {
  protected String gazda, fisier;
  protected int port;
  protected Writer iesire;
  protected BufferedReader intrare;

  /** Constructorul clasei */
  public ObtinePaginaWeb(String textURL) throws IOException {
    imparte(textURL);
  }

  /** Se va crea un obiect URL din care putem
      extrage gazda, portul si numele fisierului */
  protected void imparte(String textURL)
      throws MalformedURLException {
    URL url = new URL(textURL);
    gazda = url.getHost();
    port = url.getPort();
    if (port == 1) port = 80;
    fisier = url.getFile();
  }
  /** Realizeaza o conexiune cu serverul Web */
  public void conectare() throws IOException {
    // crearea unui socket pentru comunicare
    Socket socket = new Socket(gazda, port);
    // obtinerea fluxului pentru scrierea la socket
    OutputStream iesire = socket.getOutputStream();
    // stabilirea ultimelor caractere din flux
    this.iesire = new OutputStreamWriter(iesire, "latin1");
    // obtinerea fluxului pentru citirea din socket
    InputStream intrare = socket.getInputStream();
    // stabilirea ultimelor caractere din flux
    Reader temp = new InputStreamReader(intrare, "latin1");
    this.intrare = new BufferedReader(temp);
  }

  /** Trimite o cerere HTTP la serverul Web
      si afiseaza raspunsul la consola */
  protected void aduce() throws IOException {
    // formulam o cerere HTTP 1.0 catre masina de la distanta
    iesire.write("GET " + fisier + " HTTP/1.0\r\n\n");
    iesire.flush();

    // afisarea la consola a raspunsului serverului
    PrintWriter consola = new PrintWriter(System.out);
    String linie;
    while ((linie = intrare.readLine()) != null)
      consola.println(linie);
    // golirea fluxului aferent consolei
    consola.flush();
  }

  /** Inchide conexiunea cu serverul */
  protected void deconectare() throws IOException {
    intrare.close();
  }

  /** Realizeaza conectarea la server, aducerea paginii Web
    si deconectarea de la server */
  public void incarca() throws IOException {
    conectare();
    try {
      aduce();
    }
    finally {
      deconectare();
    }
  }

  /** Metoda principala */
  public static void main(String [] args) throws IOException {
    /* static FileDescriptor.in este un pointer catre fluxul
      de intrare standard. */
    Reader tastatura = new FileReader(FileDescriptor.in);
    BufferedReader buferTastatura = new BufferedReader
      (tastatura);
    // citim URL-uri pana apasam combinatia de taste CTRL+C
    while (true) {
      String textURL;
      System.out.print("Dati un URL: ");
      // daca am apasat CTRL+C iesim din bucla while
      if ((textURL = buferTastatura.readLine()) == null)
        break;
      try {
          // crearea unei noi instante
          ObtinePaginaWeb pag = new ObtinePaginaWeb(textURL);
          // afisarea continutului paginii
          pag.incarca();
        } // tratarea exceptiei
        catch (IOException ex) {
          ex.printStackTrace();
          // trecerea la iteratia urmatoare
          continue;
        }
        System.out.println(" OK ");
      }
      // afisarea unui mesaj la terminarea aplicatiei
      System.out.println("Am terminat!");
    }
  }
```

O execuție posibilă ar putea fi:

```
Dati un URL: http://localhost/salut.html
HTTP/1.1 200 OK
Date: Sat, 14 Dec 2007 17:11:27 GMT
Server: Apache/1.3.23 (Win32)
Last-Modified: Sat, 14 Dec 2007 15:51:56 GMT
ETag: "0-6a-3dfb539c"
Accept-Ranges: bytes
Content-Length: 106
Connection: close
Content-Type: text/html

<html>
<head>
  <title>Salutari</title>
</head>
<body>
Salutari de pe serverul Web.
</body>
</html>
 OK
Dati un URL: Am terminat!
```

Observăm că este afişat şi antetul mesajului HTTP primit. După acesta urmează ca delimitator un rând vid, iar apoi conţinutul paginii Web.

9.5. Apelul metodelor la distanţă

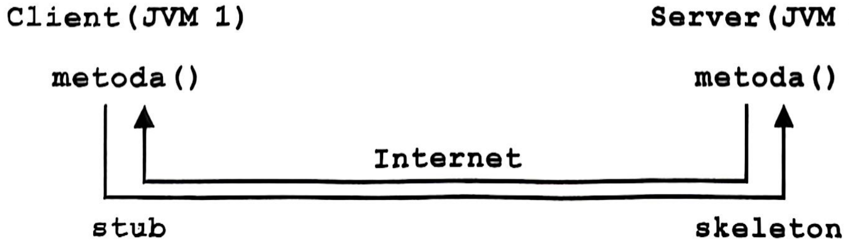
Apelul metodelor la distanţă (engl. *Remote Method Invocation* – RMI) furnizează obiectelor Java, care se află pe maşini virtuale diferite, posibilitatea să comunice utilizând apeluri de metode normale. Din moment ce gazdele (calculatoare legate la internet) pot comunica prin protocolul TCP/IP, înseamnă că aplicaţiile de reţea pot fi dezvoltate fără fluxuri şi socketuri. Aceasta permite programatorului să evite protocoalele de comunicare complexe dintre aplicaţii şi să adopte, astfel, un stil bazat pe metode de nivel înalt.

Scopul implementării apelului metodelor la distanţă în Java este furnizarea unui cadru pentru a realiza o comunicare între obiecte prin intermediul metodelor, fără să conteze localizarea lor. Asta înseamnă că un client trebuie să poată accesa un server din reţea ca şi cum s-ar executa în acelaşi sistem.

Pentru crearea unei clase care va fi accesibilă de la distanţă, trebuie să definim şi să implementăm o interfaţă care stabileşte metodele disponibile (la distanţă). Parametrii şi valorile returnate de aceste metode pot fi de orice tip: transferul de date este rezolvat automat de fluxul de obiecte. Există un instrument disponibil sub JDK pentru compilarea claselor care pot fi invocate de la distanţă, numit `rmic` (engl. *remote method invocation compiler*), care generează două fişiere de cod binar Java (cu extensia `.class`) importante:

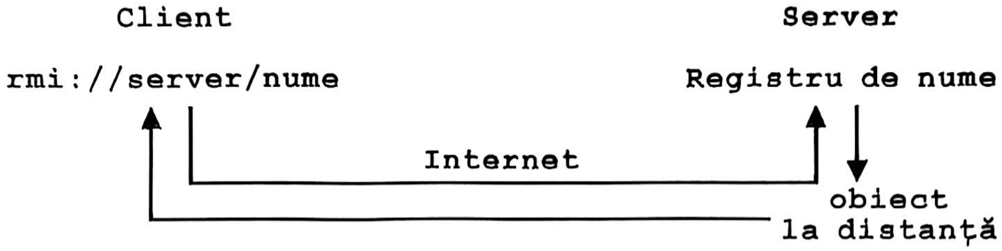
a) `stub` (buturugă, ciot),
b) `skeleton` (schelet).

`Stub`-ul este o clasă care translatează automat apelurile metodelor la distanţă în comunicaţii de reţea şi trimitere de parametri. `Skeleton`-ul este clasa care staţionează pe maşina virtuală de la distanţă şi care acceptă aceste conexiuni de reţea, pe care le translatează apoi în apeluri de metode actuale pe obiectul actual.



Utilizarea obiectelor `stub` şi `skeleton`

Pentru a putea fi utilizat, obiectul de la distanţă trebuie să fie înregistrat de un serviciu de nume care permite clienţilor să localizeze respectivul obiect. Clientul se conectează la un registru de nume şi cere o referinţă la un serviciu înregistrat sub un nume dat (de exemplu, Server RMI). Registrul de nume întoarce apoi o referinţă la distanţă la un obiect listat sub acest nume.



Înregistrarea serviciilor

În arhitectura RMI, această referință include gazda unde obiectul la distanță rulează, portul la care ascultă şi identificatorul RMI al obiectului intern. Programatorul primeşte un `stub` care implementează interfaţa cerută şi translatează automat apelurile metodei în apeluri la distanță pentru obiectul real.

După ce clientul a obținut o referință la distanță, acesta poate proceda la apelul metodelor pentru obiectul de la distanță. Cadrul RMI are grijă de toate comunicaţiile la nivel reţea. Clientul este capabil să realizeze apeluri de metode ale obiectului de la distanță ca şi cum ar face apeluri la un obiect local. Cadrul RMI include chiar şi suport pentru returnarea excepţiilor de la distanță şi pentru colectarea „gunoiului" distribuit, astfel încât obiectele de la distanță nu vor fi colectate atâta timp cât referinţele de la distanță rămân.

Există impresia că se apelează direct obiectul de la distanță de către client. Apelul este trimis de fapt `stub`-ului, care rezolvă toate detaliile de comunicare, transmiţând apelul şi parametrii către `skeleton`. Acesta face atunci un apel al metodei actuale pentru obiectul de la distanță. Valoarea returnată este în final trimisă înapoi de la `skeleton` la `stub`, care apoi returnează rezultatul clientului. Excepţiile sunt trimise înapoi, ca de altfel şi rezultatele corecte.

Dacă un obiect de la distanță este setat corect, atunci toată această muncă este transparentă pentru obiectul de la distanță şi pentru client. Nu toate metodele obiectului de la distanță pot fi apelate, ci doar acele metode declarate în interfaţa de la distanță. Cu aceste diferenţe, cadrul RMI este un mecanism extrem de puternic, deoarece el permite dezvoltarea aplicaţiilor de reţea ca şi cum ar fi un apel de metodă obişnuit.

9.5.1. Obiectul de la distanță

Un obiect de la distanță este un obiect care a fost setat să accepte apeluri de metode de la un alt obiect care rulează într-o maşină virtuală Java la distanță. Aceasta necesită două etape:

- declararea unei interfeţe care descrie metodele obiectului;
- implementarea acestei interfeţe.

Această interfaţă trebuie să extindă interfaţa `java.rmi.Remote`, care este doar un indicator (engl. *flag*) ce identifică interfeţele accesibile la distanță. Interfaţa `Remote` este superinterfaţa tuturor interfeţelor de la distanță. O interfaţă la distanță descrie metodele pe care le conţine un obiect la distanță. Acestea sunt doar metode care pot fi accesate prin RMI de un client. Interfaţa `Remote` este doar o interfaţă de marcare şi, astfel, nu are metode. Toţi parametrii şi rezultatul întors al unei metode la distanţă trebuie să fie serializabile de fluxul de obiecte.

Când un client RMI accesează un obiect la distanţă, acesta va putea accesa obiectul doar prin una sau mai multe interfeţe la distanţă. Beneficiul este că implementarea este separată de interfaţa publică.

Exemplul 9.5.1. Declararea unei interfeţe la distanţă:

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface InterfataLaDistanta extends Remote {
  public int metodaLaDistanta(String s) throws RemoteException;
}
```

În exemplul de mai sus, declarăm o interfaţă, care poate fi implementată de orice obiect la distanţă, care declară o singură metodă accesibilă la distanţă, numită `metodaLaDistanta()`, cu un singur parametru de tip `String` şi care întoarce un întreg. Toate metodele implementate într-o interfaţă la distanţă trebuie să declare că pot arunca o excepţie de tip `java.rmi.RemoteException`.

Setarea obiectului de la distanţă trebuie să importe pachetul `java.rmi.server`, să extindă `RemoteObject` (sau o subclasă a acestuia) şi să implementeze fiecare dintre interfeţele de la distanţă. De obicei, se extinde subclasa `UnicastRemoteObject` care furnizează o implementare a comportărilor unui obiect obişnuit la distanţă.

Exemplul 9.5.2. O implementare (minimală) a interfeţei declarate în exemplul 9.5.1:

```
import java.rmi.*;
import java.rmi.server.*;

public class ImplementareLaDistanta extends UnicastRemoteObject
    implements InterfataLaDistanta {
   /** constructor fara parametri */
   public ImplementareLaDistanta() throws RemoteException {}

   /** implementarea metodei din interfata */
   public int metodaLaDistanta(String s) throws RemoteException {
      // returneaza lungimea sirului de caractere
      return s.length;
   }
}
```

Clientul `stub` şi obiectul de la distanţă `skeleton` conţin tot codul implicat în detalii de serializare a apelului metodei la distanţă. Pentru serializare se utilizează clasa `ObjectOutputStream`; trimiterile se realizează prin conexiuni socket TCP/IP, iar pentru deserializare se recurge la clasa `ObjectInputStream`. Trimiterea obiectelor are loc întotdeauna doar prin valoare.

`RemoteException` este superclasa tuturor excepţiilor care pot apărea într-o execuţie RMI. Această excepţie este aruncată când eşuează un apel de metodă la distanţă. Toate metodele dintr-o interfaţă la distanţă trebuie să declare faptul că pot arunca acest tip de excepţie.

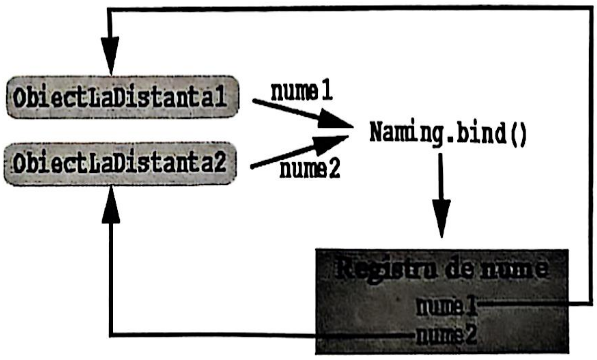
9.5.2. Registrul de nume

Clienţii pot apela metode ale obiectelor la distanţă doar dacă acestea au o referinţă la distanţă către obiect. Cadrul RMI furnizează un registru de nume pentru obţinerea acestor referinţe.

Un registru de nume poate fi pornit manual prin comanda `rmiregistry` sau programatic cu ajutorul metodelor din clasa `LocateRegistry`.

Obiectele de la distanţă pot fi înregistrate utilizând clasa `java.rmi.Naming` care foloseşte o schemă de numire asemănătoare cu cea de la URL-uri. Un registru de nume este el însuşi un client simplu RMI, deci acesta trebuie să aibă acces la fişierele `stub` pentru toate clasele care sunt înregistrate; adică acestea trebuie să fie stocate într-un director care se regăseşte în `CLASSPATH`.

Când un obiect se înregistrează cu registrul de nume, acesta specifică un nume sub care trebuie referit. Clienţii pot utiliza clasa `java.rmi.Naming` pentru a obţine o referinţă la distanţă a obiectului din registru. Obiectul este simplu identificat ca un URL care conţine numele gazdei pe care registrul de nume rulează şi numele sub care obiectul este referit.



Înregistrarea programatică a serviciilor

9.5.3. Exemplu de utilizare a apelurilor la distanţă

În această secţiune prezentăm o aplicaţie client/server care permite clienţilor să afle data şi ora de la server, utilizând apeluri de metode la distanţă. Paşii creării aplicaţiei sunt:

1. definirea interfeţei la distanţă (este interfaţa prin care clienţii de la distanţă vor accesa serverul);
2. implementarea interfeţei de la distanţă. Apelurile metodelor de la distanţă ale clientului vor fi realizate prin intermediul acestei implementări. Acestea trebuie înregistrate într-un registru de nume pentru a putea fi localizate la distanţă;

3. generarea `stub`-ului şi `skeleton`-ului utilizând comanda `rmic`;
4. scrierea unui client care localizează serverul într-un registru de nume şi apoi apelarea metodelor la distanţă;
5. pornirea registrului de nume folosind comanda `rmiregistry`. Serverul poate porni şi el un registru;
6. pornirea serverului;
7. execuţia clientului.

Pentru apelarea metodelor aflate la distanţă, vom urmări etapele mai sus menţionate.

1. Declararea interfeţei la distanţă (`ServerDeDate`):

```
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.Date;

public interface ServerDeDate extends Remote {
  /** metoda pentru aflarea datei */
  public Date obtineData() throws RemoteException;
}
```

2. Definirea clasei care implementează interfaţa de mai sus:

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Date;

public class ImplementareServerDeDate extends
    UnicastRemoteObject implements ServerDeDate {
  /** declararea unui constructor fara parametri */
  public ImplementareServerDeDate() throws RemoteException

  /** implementarea metodei pentru aflarea datei */
  public Date obtineData() {
    return new Date();
  }
  public static void main(String [] args) throws Exception {
    // obtinerea unei instante a clasei curente
    ImplementareServerDeDate serverDeDate = new
          ImplementareServerDeDate();
    // inregistrarea obiectului la distanta
    Naming.bind("Date", serverDeDate);
  }
}
```

3. Generarea `stub`-ului şi `skeleton`-ului cu ajutorul comenzii:

```
rmic ImplementareServerDeDate
```

Se vor genera două clase: `ImplementareServerDeDate_Stub.class` şi `ImplementareServerDeDate_Skel.class`.

4. Implementarea clasei corespunzătoare clientului:

```
import java.rmi.Naming;
import java.util.Date;

public class DataClient {
  public static void main(String args[]) throws Exception {
    if (args.length != 1)
      throw new IllegalArgumentException
        ("Sintaxa: ClientData <numeGazda>");
    // folosim metoda lookup() din clasa Naming
    // pentru colectarea unei referinte
    // a implementarii ServerDeDate. Un URL cu acest
    // camp protocol este folosit pentru numele
    // obiectului; partea cu rmi poate fi omisa deci
    // putem pune doar ca URL //server/obiect
    // obtinerea unei referinte la un obiect aflat la distanta
    ServerDeDate serverData = (ServerDeDate) Naming.lookup
      ("rmi://" + args[0] + "/Date");
    // apelarea unei metode la distanta
    Date data = serverData.obtineData();
    // afisarea rezultatului
    System.out.println(data);
  }
}
```

5. Pornirea registrului de pe maşina-server utilizând comanda:

```
rmiregistry
```

Acest registru este acum accesibil clienţilor de la distanţă pe un port standard (bine cunoscut). Comanda este executată din directorul unde sunt plasate fişierele `stub`.

6. Pornirea serverului se realizează astfel:

```
java ImplementareServerDeDate
```

7. Execuţia clientului (pe maşina locală):

```
java DataClient localhost
```

sau (echivalent):

```
java ClientData 127.0.0.1
```

Continuăm cu explicaţii referitoare la execuţia aplicaţiei de mai sus. Clasa `Naming` se conectează la regiştrii care rulează pe gazda specificată şi apoi caută obiectul care se potriveşte cu numele indicat. Obiectul returnat de acest apel este o referinţă la distanţă (de fapt, o instanţă a unui `stub`). Astfel, trebuie să convertim rezultatul la interfaţa de la distanţă aşteptată, şi nu numele clasei implementate la distanţă. Doar metodele declarate în interfeţele de la distanţă sunt accesibile prin RMI.

Când s-a obţinut o referinţă la distanţă a obiectului, putem apela metode declarate în interfeţele lor de la distanţă. În acest caz, facem un apel la metoda `obtineData()` şi afişăm la consolă răspunsul. Când realizăm acest apel de metodă, `stub`-ul local se conectează la `skeleton`-ul de la distanţă, care apelează metoda obiectului de la distanţă, obţine rezultatul şi prin intermediul fluxurilor se va trimite înapoi la `stub`.

9.5.4. Localizarea fişierelor RMI. Pachete necesare

Din punct de vedere al unei aplicaţii RMI, există cinci tipuri de fişiere `.class`: cu implementarea clientului, a serverului, interfaţa de la distanţă, clasele `stub` şi `skeleton`.

Pentru distribuirea unui client RMI trebuie să includem clasele de implementare a clientului, clasele de interfaţă la distanţă şi clasele `stub`.

Pentru distribuirea unui server RMI trebuie să includem clasele de implementare a serverului, interfeţele de la distanţă, clasele `stub` şi clasele `skeleton`. Toate aceste clase sunt necesare în timpul rulării serverului şi cu toate că `stub`-ul este executat doar de client, acesta este necesar când se exportă la distanţă.

Anumite aplicaţii RMI nu separă aşa de categoric partea de client şi cea de server. Este posibil ca şi serverul să poată executa anumite metode la distanţă (nu numai clientul).

Principalele pachete Java care sunt utilizate de aplicaţiile ce invocă metode la distanţă sunt:

- `java.rmi` – oferă suport pentru partea de client RMI; include interfeţele prin care sunt accesate obiectele la distanţă şi un mecanism pentru localizarea serviciilor RMI pe o maşină la distanţă;
- `java.rmi.server` – conţine clase referitoare la partea de server RMI; include clase pentru lucrul cu servicii RMI, pentru rezolvarea cererilor TCP/IP şi HTTP;
- `java.rmi.registry` – pune la dispoziţie clase pentru lucru cu regiştri de nume RMI (creare, localizare şi manipulare la distanţă);
- `java.rmi.dgc` – conţine clase referitoare la eliberarea memoriei (engl. *garbage collector*); serverele RMI păstrează automat un număr de referinţe la distanţă active pe care le servesc şi le pot închide, dacă nu mai sunt referite;
- `java.rmi.activation` – oferă suport pentru salvarea stărilor obiectelor la distanţă atunci când sunt neutilizate şi respectiv restaurarea acestora când apar cereri. Serverele RMI nu trebuie neapărat să ruleze mereu; ele pot fi activate de un daemon de activare doar dacă soseşte o cerere şi sunt ţinute într-o stare pasivă serializată.

9.5.5. Clasa `Naming`

Clasa `Naming` face parte din pachetul `java.rmi` şi oferă posibilitatea manipulării regiştrilor de nume RMI, localizării şi înregistrării obiectelor. În implementarea sa, această clasă utilizează interfaţa `Registry` şi clasa `LocateRegistry` şi furnizează metode statice şi notaţii URL pentru adresarea obiectelor. De regulă, un client poate utiliza această clasă pentru localizarea unui obiect la distanţă, iar un server, pentru înregistrare.

Un URL pentru RMI obişnuit are forma `rmi://numeGazda:port/serviciu`. Adresa identifică obiectul la distanţă, numit serviciu, memorat într-un registru de nume care rulează la portul specificat de pe gazda indicată. De exemplu, adresa `rmi://dil.infoiasi.ro:2500/java/exemple` se referă la un obiect la distanţă înregistrat sub numele de `java/exemple` într-un registru de nume care ascultă la portul `2500` al maşinii `dil.infoiasi.ro`.

Toate metodele clasei `Naming` sunt statice şi nu posedă constructori expliciţi. Metodele acestei clase sunt descrise în tabelul următor:

Prototipul metodei	Descrierea metodei
`Remote` **`lookup`**`(String adresa) throws MalformedURLException, RemoteException, NotBoundException`	Se conectează la registrul de nume specificat şi extrage o referinţă a obiectului de la distanţă înregistrat sub numele de serviciu indicat.
`void` **`bind`**`(String adresa, Remote obiect) throws MalformedURLException, RemoteException, AlreadyBoundException`	Stabileşte numele serviciului (`adresa`) pentru obiectul specificat.
`void` **`rebind`**`(String adresa, Remote obiect) throws MalformedURLException, RemoteException`	Înlocuieşte obiectul la care se referă serviciul cu obiectul specificat.
`void` **`unbind`**`(String adresa, Remote obiect) throws MalformedURLException, RemoteException`	Elimină serviciul specificat.
`String []` **`list`**`(String adresa) throws MalformedURLException, RemoteException`	Returnează lista tuturor serviciilor disponibile la distanţă înregistrate în registrul de nume la adresa specificată.

Excepţia `MalformedURLException` apare atunci când adresa RMI nu este validă, `RemoteException`, dacă regiştrii de nume nu pot fi consultaţi, `NotBoundException`, dacă serviciul nu este încă legat (setat), iar `AlreadyBoundException` când se încearcă stabilirea unui nume de serviciu care există deja.

9.5.6. Clasa `LocateRegistry`

Clasa `LocateRegistry`, din pachetul `java.rmi.registry`, este utilă pentru controlul regiştrilor de nume. Aceasta furnizează metode pentru obţinerea referinţelor la distanţă către regiştrii de nume.

Toate metodele clasei sunt statice şi pot arunca excepţia `RemoteException`. Acestea returnează obiecte care pot fi manipulate prin intermediul interfeţei `Registry`. Metodele clasei `LocateRegistry` sunt:

Prototipul metodei	Descrierea metodei
`Registry` **`getRegistry`**`() throws RemoteException`	Returnează o referinţă la distanţă pentru registrul de nume de pe maşina locală la portul implicit.
`Registry` **`getRegistry`**`(int port) throws RemoteException`	Întoarce o referinţă la distanţă pentru registrul de nume de la portul specificat al maşinii locale.
`Registry` **`getRegistry`**`(String gazda) throws RemoteException`	Returnează o referinţă la distanţă pentru registrul de nume aflat pe maşina-gazdă specificată, la portul implicit.
`Registry` **`getRegistry`**`(String gazda, int port) throws RemoteException`	Întoarce o referinţă la distanţă pentru registrul de nume de la portul şi gazda indicate.
`Registry` **`createRegistry`**`(int port) throws RemoteException`	Creează şi porneşte un registru de nume pe maşina locală la portul specificat. Returnează o referinţă directă a registrului rezultat, nu o referinţă la distanţă, cum returnează celelalte metode.

Exemplul 9.5.3. Căutarea unui registru de nume utilizând clasa `LocateRegistry`:

```
Registry registru = LocateRegistry.getRegistry("gazda", 1234);
registru.rebind("Serviciu", serviciu);
```

Crearea unui registru de nume local este simplă:

```
Registry registru = LocateRegistry.createRegistry(1234);
registru.bind("Serviciu", serviciu);
```

9.5.7. Interfaţa `Registry`

Interfaţă `Registry` din pachetul `java.rmi.registry` descrie partea publică a registrului de nume RMI. Acesta este implementat ca un obiect la distanţă; clasa `LocateRegistry` va returna referinţe la distanţă pentru implementarea JDK a interfeţei.

Interfaţa `Registry` declară o singură constantă statică:

- `int REGISTRY_PORT;`
  care este numărul portului implicit, unde ascultă registrul de nume (iniţial, este valoarea `1099`).

Spre deosebire de clasa `Naming`, când specificăm un nume de serviciu prin intermediul acestei interfeţe, se poate preciza doar numele serviciului, nu şi adresa registrului de nume. Metodele clasei `Naming` sunt de fapt implementate prin apeluri ale clasei `LocateRegistry`, urmate de apeluri la distanţă cu ajutorul interfeţei `Registry`.

Metodele interfeţei `Registry` sunt următoarele:

Prototipul metodei	Descrierea metodei
`Remote` **`lookup`**`(String nume) throws RemoteException,NotBoundException, AccessException`	Returnează o referinţă la distanţă către serviciul înregistrat în registrul de nume sub numele specificat.
`void` **`bind`**`(String nume, Remote obiect) throws RemoteException, AlreadyBoundException, AccessException`	Stochează o referinţă a obiectului de la distanţă specificat în registrul sub numele `nume`.
`void` **`rebind`**`(String nume, Remote obiect) throws RemoteException, AccessException`	Modifică obiectul de la distanţă cu cel specificat în registrul de nume, suprascriind orice intrare existentă.
`void` **`unbind`**`(String nume) throws RemoteException,NotBoundException, AccessException`	Şterge orice serviciu înregistrat sub numele nume din registru de nume.
`String []` **`list`**`() throws RemoteException, AccessException`	Returnează lista completă a tuturor serviciilor disponibile din acest serviciu. Doar numele serviciului este returnat; nu este inclusă nici o informaţie despre adresă.

9.5.8. Clasa `RemoteObject`

Clasa `java.rmi.server.RemoteObject` este superclasa tuturor obiectelor la distanţă. Aceasta suprascrie metodele `hashCode()`, `equals()` şi `toString()` pentru a reflecta semanticile corecte ale obiectului la distanţă. Clasa implementează interfaţa `Serializable` pentru descrierea corectă a serializării obiectelor de la distanţă. Implementează, de asemenea, interfaţa `Remote`. Această clasă nu posedă alte metode importante în afară de cele menţionate mai sus.

9.5.9. Obiecte la distanţă ca parametri

Cu toate că RMI utilizează fluxuri de obiecte pentru trimiterea parametrilor către un apel de metodă la distanţă, acesta, de fapt, foloseşte subclase specializate care suprascriu serializarea implicită pentru obiecte la distanţă.

Când se trimite o implementare a unui obiect la distanţă (a subclasei `RemoteObject`), ca parametru către un apel de metodă la distanţă, se trimite, de fapt, o referinţă la distanţă către o implementare de obiect la distanţă şi nu o copie a acestuia (ca în cazul obiectelor locale). Aşadar, când metoda de la distanţă apelează metode ale unor obiecte pe care le primeşte, atunci au loc mai multe apeluri RMI pentru implementarea obiectului de la distanţă original.

Consecinţa este că nu se poate trimite un obiect de la distanţă ca parametru pentru un apel al unei metode la distanţă şi să ne aşteptăm ca destinatarul să primească o instanţă a clasei obiectului de la distanţă actual. În schimb, destinatarul va primi o instanţă a clasei `stub` care conţine o referinţă de la distanţă către implementarea obiectului actual.

Exemplul 9.5.4. Transmiterea unui obiect la distanţă ca parametru pentru o metodă aflată la distanţă:

```
public class OImplementare extends UnicastRemoteObject
    implements InterfataNoastra {
   // . . .
  public static void main(String [] args) throws Exception {
    OImplementare ob1 = new OImplementare();
    AltaDistanta ob2 = (AltaDistanta) Naming.lookup(
      "//server/ob2");
    ob2.apelare(ob1);
  }
}
```

În fragmentul de cod de mai sus am creat un obiect la distanţă local, `ob1`, pe care îl trimitem ca parametru unei metode a altui obiect la distanţă, `ob2`. După transferul de date RMI, metoda `apelare()` nu va primi un obiect de tip `OImplementare`, ci un obiect de tip `OImplementare_Stub` care implementează interfaţa `InterfataNoastra` şi utilizează metodele definite de aceasta.

Deci, este greşit să definim o metodă la distanţă care primeşte ca parametru un obiect de la distanţă actual definit în clasa implementată. Se pot defini metode la distanţă care primesc ca parametri fie obiecte locale, fie interfeţe la distanţă.

Sunt posibile următoarele declaraţii pentru metoda `apelare()`:

- `public void apelare(Object obiect) throws RemoteException;`
  Signatura acestei metode este validă, deoarece se va transmite un obiect de tip `OImplementare_Stub`, care este subclasă a lui `Object`;
- `public void apelare(InterfataNoastra distanta) throws RemoteException;`
  Această variantă este validă, deoarece `OImplementare_Stub` implementează `InterfataNoastra`.

Următoarea declaraţie este nevalidă:

- `public void apelare(OImplementare obiect) throws RemoteException;`
  Nu se obţine eroare la compilare. Cu toate acestea, cadrul RMI va converti o instanţă a clasei `OImplementare` într-o instanţă pentru `OImplementare_Stub` în timpul serializării. Clasa `stub` nu este o subclasă a `OImplementare_Stub` şi la execuţie vom obţine o excepţie RMI.

9.5.10. Serializarea unui obiect la distanţă

Când serializăm un obiect la distanţă prin intermediul fluxurilor de obiecte locale, se va serializa un obiect la distanţă valid. La deserializare, obiectul este automat restaurat. Utilizând acest mecanism, putem serializa un obiect la distanţă pe disc şi apoi să-l deserializăm fie cu aceeaşi maşină virtuală, fie cu ajutorul alteia. În urma acestui proces, va fi creat un obiect la distanţă valid.

Exemplul 9.5.5. Serializarea unui obiect la distanţă.

```
// ObjectOutputStream obiectDeIesire;
OImplementare ob1 = new OImplementare();
obiectDeIesire.writeObject(ob1);
```

Instanţa lui `OImplementare` este serializată, utilizând procesul de serializare obişnuit.

Exemplul 9.5.6. Restaurarea unui obiect la distanţă.

```
// ObjectInputStream obiectDeIntrare;
OImplementare ob2 = (OImplementare) obiectDeIesire.readObject();
```

Obţinem, astfel, un duplicat al obiectului de la distanţă original. Acesta este un obiect la distanţă complet activat care poate fi înregistrat într-un registru de nume şi manipulat exact ca o instanţă originală.

9.5.11. Clasa `RemoteServer`

Clasa `java.rmi.server.RemoteServer` furnizează funcţiile de bază ale obiectelor la distanţă care vor fi implementate ca servere (adică obiecte la distanţă care vor deschide socketuri pentru acceptarea de apeluri-client).

Clasa `RemoteServer` extinde clasa `RemoteObject` şi posedă în plus următoarele metode (toate sunt publice şi statice):

Prototipul metodei	Descrierea metodei
`String` **`getClientHost`**`() throws ServerNotActive-Exception`	Când este apelată dintr-o implementare a unei metode la distanţă, returnează adresa maşinii-client, altfel aruncă o excepţie de tip `ServerNotActiveException`.
`void` **`setLog`**`(OutputStream iesire)`	RMI furnizează posibilitatea de jurnalizare a tuturor apelurilor RMI. Această metodă stabileşte care este fluxul pentru înregistrarea apelurilor.
`PrintStream` **`getLog`**`()`	Returnează fluxul pentru jurnalizare.

Exemplul 9.5.7. Apelul de mai jos permite înregistrarea tuturor apelurilor RMI în fişierul `rmi.log`.

```
RemoteServer.setLog(new FileOutputStream("rmi.log"));
```

Apelul de mai jos va închide fluxul pentru jurnalizare. El nu închide fluxul original.

```
RemoteServer.setLog(null);
```

9.5.12. Clasa `RemoteStub`

Clasa `java.rmi.server.RemoteStub` este superclasa tuturor claselor `stub` de la distanţă. Când se rulează `rmic`, fişierul `stub` produs este o subclasă a lui `RemoteStub`. Acesta implementează aceleaşi interfeţe ca şi clasa la distanţă originală, dar furnizează toate metodele la distanţă pentru implementarea obiectului de la distanţă actual (prin conectare şi comunicare cu `skeleton`-ul). Când se caută o referinţă la distanţă într-un registru de nume, se primeşte, de fapt, o instanţă a unei clase `stub`.

Clasa `RemoteStub` nu furnizează metode pentru uz particular.

9.5.13. Interfaţa `Unreferenced`

Interfaţa `Unreferenced` este utilă pentru obiecte la distanţă care doresc să fie notificate de fiecare dată când nu există referinţe la distanţă înapoi către ele. Pentru implementarea colectorului de gunoi la distanţă, cadrul RMI menţine automat un contor al numărului de referinţe care există către fiecare obiect la distanţă. Această interfaţă face parte din pachetul `java.rmi.server`.

Există doar o metodă definită:

- `public void unreferenced()`

  Aceasta va fi apelată automat de fiecare dată când nu mai sunt referinţe la respectivul obiect (contorul pentru numărul de referinţe ajunge la valoarea zero). Contorul de referinţă pentru obiectele aflate la distanţă este independent de contorul de referinţă local. Mecanismul este, de obicei, utilizat pentru dezactivare manuală a obiectelor, când acestea nu mai sunt referite.

O intrare într-un registru de nume este, de fapt, o referinţă la distanţă către obiectul înregistrat, deci atât timp cât obiectul este memorat într-un registru de nume, acesta nu va fi nereferit.

Colectarea gunoiului distribuit implică notificarea periodică a referinţelor la distanţă ale unui obiect la distanţă care sunt încă active. Perioada acestei notificări este chiar mare (implicit, 10 minute).

9.5.14. Clasa `RMISocketFactory`

Un obiect de tip `RMISocketFactory` este utilizat de cadrul RMI pentru obţinerea socketului client sau server prin intermediul căruia au loc apeluri la distanţă. Clasa `RMISocketFactory` este foarte importantă, deoarece permite stabilirea parametrilor impliciţi ai socketurilor utilizate în aplicaţiile RMI.

Când cadrul RMI doreşte să se conecteze la un obiect la distanţă, încearcă mai întâi să creeze o conexiune TCP directă către obiect. În acest caz, cererea poate fi de tipul următor:

```
raw://server:portObiectRMI
```

Din spatele unui firewall, o astfel de conexiune poate să nu fie permisă. În situaţia unui eşec, se încearcă o conexiune HTTP la un port la care obiectul de la distanţă ascultă. Obiectele de la distanţă furnizează un suport predefinit pentru transferul direct dintre date şi transferul datelor prin intermediul protocolului HTTP. Astfel, cadrul RMI se va conecta la proxy-ul local (dacă proxy-ul există) pentru procurarea obiectului de la distanţă. Se va realiza o cerere de forma:

```
POST http://server:portObiectRMI/data . . .
```

Dacă firewall-ul sau proxy-ul permit doar traficul pe portul 80 al siturilor de la distanţă, atunci se va realiza o conexiune cu un program CGI (*Common Gateway Interface*) de pe serverul Web, care va dirija cererea direct către obiectul de la distanţă. În acest caz, cadrul RMI va trimite o cerere de forma:

```
POST http://server:80/cgibin/javarmi.cgi/data . . .
```

Clasa `RMISocketFactory` este abstractă şi deci nu îi putem crea instanţe. Metodele sale sunt descrise în tabelul de mai jos:

Prototipul metodei	Descrierea metodei
`static void` **`setSocketFactory`** `(RMISocketFactory fabrica) throws IOException`	Instalează o nouă *fabrică* de socketuri. Noua clasă va fi responsabilă de crearea tuturor socketurilor viitoare în această maşină virtuală.
`static RMISocketFactory` **`getSocketFactory`**`()`	Returnează fabrica de socketuri curentă, dacă există, sau `null`, în caz contrar (când este utilizată cea implicită).
`static RMISocketFactory` **`getDefaultSocketFactory`**`()`	Întoarce fabrica de socketuri implicită.
`static void` **`setFailureHandler`** `(RMIFailureHandler rezolvitorEsec)`	Stabileşte un rezolvitor de eşec RMI. Acesta este apelat atunci când crearea unui socket RMI se încheie cu un eşec.
`static RMIFailureHandler` **`getFailureHandler`**`()`	Returnează rezolvitorul de eşec curent.
`abstract Socket` **`createSocket`** `(String gazda, int port) throws IOException`	Returnează un nou `Socket` conectat la gazda şi portul specificate. Orice excepţie poate fi aruncată şi trimisă mai departe sau poate fi rezolvată intern.
`abstract ServerSocket` **`createServerSocket`**`(int port) throws IOException`	Întoarce un nou `ServerSocket` care ascultă la portul specificat. Excepţiile pot fi tratate sau trimise mai departe.

Se poate stabili o singură dată fabrica de socketuri; metoda `setSocketFactory()` va arunca o excepţie `IOException`, dacă este deja definită o fabrică. Această metodă este controlată şi de managerul de securitate, deci codurile nesigure nu vor putea înregistra o fabrică socket.

9.5.15. Interfaţa `RMIFailureHandler`

Interfaţă descrie un rezolvitor de eşec RMI. Ea încearcă să rezolve cauza unei excepţii apărute la crearea unui socket RMI. Dacă reuşeşte, cadrul RMI poate încerca o nouă conectare.

Există o singură metodă declarată în această interfaţă:

- `boolean` **`failure`**`(Exception ex);`
  care este apelată când crearea socketului RMI a eşuat, iar ca parametru este trimisă excepţia apărută. O implementare ar trebui să încerce să rezolve problema şi să returneze `true`, pentru a se mai încerca crearea socketului de către cadrul RMI.

9.5.16. Exemplu de aplicaţie RMI

Vom prezenta o implementare bazată pe RMI a unui sistem de discuţii (engl. *chat*) de tip client/server. Serverul este dat de clasa `ImplementareServerChatRMI`, iar o instanţă a acesteia este un obiect la distanţă care menţine o listă internă a tuturor mesajelor primite de la clienţi. Clasa pentru server implementează interfaţa `ServerChatRMI`, prin care clienţii pot adăuga mesaje noi şi pot interoga lista curentă de mesaje.

Clientul, `ClientChatRMI`, este o aplicaţie simplă; acesta trimite mesaje către server într-o manieră simplă şi posedă un fir de execuţie care interoghează periodic serverul pentru o copie actualizată a sesiunii *chat*.

Interfaţa `ServerChatRMI` stabileşte metodele care vor fi accesibile la distanţă:

```
import java.rmi.*;

/** Interfata pentru obiectul de la distanta */
public interface ServerChatRMI extends Remote {
  /** Stabileste numele pentru registru */
  public static final String NUME_REGISTRU = "ServerChat";

 /** Defineste prototipului metodei pentru preluarea mesajelor
    noi */
  public abstract String[] obtineMesaje(int index)
    throws RemoteException;

  /** Defineste prototipului metodei pentru adaugarea unui
    mesaj */
  public abstract void adaugaMesaj(String mesaj)
```

```
     throws RemoteException;
}
```

Codul clasei `ImplementareServerChatRMI` este următorul:

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.util.*;

public class ImplementareServerChatRMI extends UnicastRemoteObject
    implements ServerChatRMI {
  // Declararea datelor membre
  protected Vector<String> mesaje; //tabloul de mesaje

  /** Declararea constructorului fara parametri */
  public ImplementareServerChatRMI() throws RemoteException {
    // Crearea vecţorului pentru mesaje
    mesaje = new Vector<String>();
  }

  /** Obtine mesajele noi */
  public String[] obtineMesaje(int index) {
    // Aflarea numarului total de mesaje
    int dimensiune = mesaje.size();
    // Crearea tabloului pentru mesajele noi
    String[] actualizat = new String[dimensiune - index];
    // Preluarea mesajelor noi
    for (int i = 0; i < dimensiune - index; ++ i)
      actualizat[i] = mesaje.elementAt(index + i);
    // Returnarea rezultatului
    return actualizat;
  }

  /** Adauga un mesaj */
  public void adaugaMesaj(String mesaj) {
    // adaugarea mesajului in vectorul de mesaje
    mesaje.addElement(mesaj);
  }

  /** Metoda principala */
  public static void main(String [] args) throws
  RemoteException {
    // Obtinerea unei instante a clasei pentru server
    ImplementareServerChatRMI serverChat = new
        ImplementareServerChatRMI();
      // inregistrarea obiectului la distanta
      Registry registru = LocateRegistry.getRegistry();
      registru.rebind(NUME_REGISTRU, serverChat);
   }
}
```

Pentru a putea executa serverul, trebuie compilat programul de mai sus (cu ajutorul comenzii `rmic ImplementareServerChatRMI`), rezultând fişierele *`stub`* şi *`skeleton`*, apoi să pornim registrul de nume (utilizând comanda `rmiregistry`). Pornirea serverului se realizează cu ajutorul comenzii: `java ImplementareServerChatRMI`.

Aplicaţia-client este următoarea:

```
import java.awt.*;
import java.awt.event.*;
import java.rmi.*;
import java.rmi.registry.*;

public class ClientChatRMI implements Runnable, ActionListener
  // definirea datelor membre
  protected static final int PAUZA_ACTUALIZARE = 100;
  protected String gazda;
  protected Frame cadru;
  protected TextField intrare;
  protected TextArea iesire;
  protected ServerChatRMI server;
  protected Thread actualizator;

  /** Definirea constructorului */
  public ClientChatRMI (String gazda) {
    this.gazda = gazda;
    // crearea unei ferestre
    cadru = new Frame("ClientChatRMI[" + gazda + "]");
    // crearea unui element textarea pentru afisarea mesajelor
    cadru.add(iesire = new TextArea(), "Center");
    iesire.setEditable(false);
    // crearea unui camp text pentru inserarea mesajelor noi
    cadru.add(intrare = new TextField(), "South");
    /// stabilirea ascultatorilor
    intrare.addActionListener(this);
    // urmeaza declaratia unei clase interioare
    cadru.addWindowListener(new WindowAdapter() {
       public void windowOpened(WindowEvent ev) {
         intrare.requestFocus();
       }
       public void windowClosing(WindowEvent ev) {
         opreste();
```

```
        System.exit(0);
      }
  }); // sfarsitul apelului metodei addWindowListener
  cadru.pack();
} // sfarsitul constructorului ClientChatRMI

/** Metoda pentru pornirea chatului */
public synchronized void porneste() throws RemoteException,
  NotBoundException {
  if (actualizator == null) {
    // obtinerea registrului
    Registry registru = LocateRegistry.getRegistry(gazda);
    // obtinerea unei instante la obiectul la distanta
    server = (ServerChatRMI) registru.lookup(
       ServerChatRMI.NUME_REGISTRU);
    // crearea unui fir de executie
    actualizator = new Thread(this);
    actualizator.start();
    // vizualizarea ferestrei
    cadru.setVisible(true);
  }
}

// metode pentru oprirea clientului
public synchronized void opreste() {
  // intrerupem activitatea firului de executie
  if (actualizator != null) {
    actualizator.interrupt();
    actualizator = null;
    server = null;
  }
  // ascundem fereastra
  cadru.setVisible(false);
}

// executia firului de executie
public void run() {
  try {
    // incepem cu primul mesaj
    int index = 0;
    // citim mesaje atat timp cat nu este intrerupt firul
    // de executie
    while (!Thread.interrupted()) {
      // obtinem mesajele noi
      String [] mesaje = server.obtineMesaje(index);
```

```
        // afisam mesajele in campul de tip textarea
        int n = mesaje.length;
        for (int i = 0; i < n; ++ i)
          iesire.append(mesaje[i] + "\n");
        index += n;
        // firul de executie va astepta un timp
        // pana la preluarea altor mesaje
        Thread.sleep(PAUZA_ACTUALIZARE);
      }
    }  // tratarea exceptiilor
    catch (InterruptedException ie) {}
    catch (RemoteException ex) {
      intrare.setVisible(false);
      cadru.validate();
      ex.printStackTrace();
    }
  } // sfarsitul metodei run()

  /** Metoda se apeleaza la apasarea tastei ENTER in campul
  text */
  public void actionPerformed(ActionEvent ev) {
    try {
      ServerChatRMI server = this.server;
      // daca suntem conectati trimitem mesajul
      if (server != null)
        server.adaugaMesaj(ev.getActionCommand());
      // stergem mesajul trimis
      intrare.setText("");
    } // Tratarea exceptiilor
    catch (RemoteException ex) {
      Thread tmp = actualizator;
      actualizator = null;
      if (tmp != null)
        tmp.interrupt();
      intrare.setVisible(false);
      cadru.validate();
      ex.printStackTrace();
    }
  } // de la public void actionPerformed(ActionEvent ev)

  /** metoda principala */
  public static void main(String [] args)
     throws RemoteException, NotBoundException {
    // primul parametru va fi numele masinii-server
    if (args.length != 1)
```

```
            throw new IllegalArgumentException
              ("Sintaxa: ClientChatRMI <gazda>");
        // cream o instanta a clientului
        ClientChatRMI clientChat = new ClientChatRMI(args[0]);
        // pornim chatul
        clientChat.porneste();
    }
}
```

Compilarea şi execuţia se realizează la fel ca pentru orice aplicaţie Java de sine stătătoare (obişnuită).

Rămâne în sarcina utilizatorilor să stabilească un nume pentru fiecare utilizator, pentru a putea şti cine a trimis mesajele. La anumite intervale de timp, va trebui să eliminăm mesajele vechi pentru a nu ocupa prea multe resurse.

9.6. Concluzii

Capitolul „Programarea reţelelor" a debutat cu câteva elemente introductive care au stabilit termenii de bază vehiculaţi în programarea reţelelor de calculatoare (adrese IP, porturi, socketuri, conexiuni, datagrame etc.). Apoi au fost prezentate diferenţele dintre conexiuni şi datagrame. Conexiunile sunt asemănătoare apelurilor telefonice (simultan, ambii parteneri de comunicaţie trebuie să fie disponibili), iar datagramele, sistemului de poştă obişnuită. Exemplele prezentate evidenţiază uşurinţa lucrului cu socketuri atât pentru conexiuni, cât şi pentru datagrame.

Clasa `URL` permite atât specificarea URL-urilor, cât şi obţinerea într-o manieră facilă a conţinutului resurselor referite de ele. Această facilitate reduce timpul de accesare a resursei, precum şi codul scris de programator.

O modalitate de programare a reţelelor de calculatoare e reprezentată de apelurile metodelor aflate la distanţă (pe alte maşini din reţea). De data aceasta, comunicarea parametrilor şi returnarea rezultatelor este realizată automat, iar lucrul cu obiectele aflate la distanţă este similar cu cele locale. Obiectele aflate la distanţă se pot serializa ca şi obiectele obişnuite, la restaurarea acestora neexistând probleme.

La compilarea obiectelor aflate la distanţă rezultă două fişiere de tip `.class`. Pentru a fi accesibil la distanţă, un obiect trebuie să fie înregistrat într-un serviciu de nume.

Atât invocarea metodelor la distanţă, cât şi socketurile permit realizarea de aplicaţii distribuite, execuţia nemaifiind pe un singur calculator, ci pe mai multe maşini din reţea.

9.7. Test-grilă

Întrebarea 9.7.1. Care dintre următoarele afirmaţii sunt false:

a) Comunicarea prin datagrame nu asigură primirea corectă (nedeteriorată) a mesajelor.

b) Protocolul HTTP este orientat conexiune.
c) La un port pot asculta mai multe servere TCP.
d) Doar un server TCP şi unul UDP pot asculta la acelaşi port.

Întrebarea 9.7.2. Care dintre afirmaţiile următoare sunt adevărate:
a) O aplicaţie server poate asculta la mai multe porturi.
b) Un socket poate să asculte la un singur port.
c) O maşină din reţea poate avea mai multe adrese IP.
d) O maşină din reţea poate avea un singur nume (domeniu).

Întrebarea 9.7.3. Apelurile de metode la distanţă se realizează:
a) prin conexiuni TCP;
b) prin datagrame;
c) prin intermediul protocolului HTTP;
d) cu ajutorul firewall-urilor.

Întrebarea 9.7.4. Denumirea de URL provine de la:
a) Universal Resource Location.
b) Universal Resource Locator.
c) Uniform Resource Location.
d) Uniform Resource Locator.

Întrebarea 9.7.5. Pentru a apela metodele unui obiect la distanţă trebuie să:
a) pornim un registru de nume;
b) creăm un socket pentru a stabili o conexiune cu serverul RMI;
c) realizăm o cerere pentru respectivul obiect prin intermediul socketului creat anterior;
d) obţinem o instanţă a obiectului la distanţă.

Întrebarea 9.7.6. Care dintre afirmaţiile următoare, referitoare la trimiterea obiectelor la distanţă ca parametri ai metodelor la distanţă, sunt adevărate?
a) Nu se pot trimite.
b) Se trimite o copie a lor.
c) Se trimite o referinţă la distanţă.
d) Se trimite obiectul *stub* corespunzător.
e) Se trimite obiectul *skeleton* corespunzător.

Întrebarea 9.7.7. Serializarea obiectelor la distanţă se poate realiza:
a) ca la obiectele locale;
b) ca la obiectele locale, numai că la restaurare vor apărea probleme;
c) într-un mod specific, RMI implementând interfaţa `RMISerializable`;
d) nu se pot serializa obiectele la distanţă.

9.8. Exerciţii propuse spre implementare

Exerciţiul 9.8.1. (*Calculator*) Scrieţi o aplicaţie client/server în care clientul trimite o expresie aritmetică la server şi serverul întoarce valoarea acesteia, care se afişează pe situl client.

Exerciţiul 9.8.2. Să se implementeze un obiect la distanţă cu o metodă care primeşte adresa de IP sau numele domeniului unei maşini din reţea şi returnează clasa din care face parte (din cele cinci existente: A, B, C, D, E). Pentru stabilirea clasei din care face parte va trebui să cercetăm primul număr din adresa IP. Dacă acesta are primul bit 0, atunci este de clasă A, dacă primii 2 biţi sunt 10, atunci este de clasă B, pentru 110 avem clasa C, 1110 pentru D şi 11110 pentru E.

Exerciţiul 9.8.3. Scrieţi o aplicaţie client/server, bazată pe RMI, a cărei interfaţă conţine două metode: una pentru iniţializarea unui obiect de tip Vector şi a doua pentru returnarea sumei elementelor vectorului.

Exerciţiul 9.8.4. Să se creeze o aplicaţie care permite specificarea adresei unui sit Web şi verificarea dacă acesta are „legături moarte" (legături la pagini inexistente).

Exerciţiul 9.8.5. Să se implementeze un motor de căutare pentru situri Web. Aplicaţia va avea ca intrare adresa unui sit Web şi o secvenţă de cuvinte, iar ca ieşire lista tuturor paginilor din respectivul sit care conţin secvenţa respectivă şi contextul în care apare. Eventual, se va da posibilitatea de a realiza căutări doar în cadrul paginilor din ultimul rezultat obţinut.

9.9. Proiecte propuse spre implementare

Proiectul 9.9.1. Scrieţi o aplicaţie client/server care să simuleze un sistem (grafic) de *chat*. Clientul trimite un mesaj la server şi serverul trimite la toţi utilizatorii (clienţii) acelui *chat* mesajul primit. Se va da posibilitatea de a vizualiza utilizatorii conectaţi, de a trimite mesaje individuale sau la grupuri de utilizatori, de a crea şi administra „camere" de discuţii, posibilitatea de a restricţiona accesul persoanelor cu un comportament inadecvat.

Proiectul 9.9.2. Să se conceapă un sistem de tip Napster, în care fiecare utilizator pune la dispoziţie o serie de fişiere (audio, video, componente software, programe etc.). Cu ajutorul unui serviciu de căutare, se poate depista dacă există fişierele dorite de un anumit utilizator şi persoanele care oferă respectivele fişiere. Se va da posibilitatea de a copia fişiere de la distanţă (de la alţi utilizatori).

10. Servleturi

10.1. Cuvinte-cheie

- servlet, CGI
- server Web, Tomcat

10.2. CGI (*Common Gateway Interface*)

Standard pentru interacţiunea clienţilor Web cu serverele Web, *Common Gateway Interface* se află în prezent la versiunea 1.1. Un program executabil sau script CGI este executat de către serverul Web, fie în mod explicit (apelat din cadrul paginii printr-o directivă specială), fie la preluarea informaţiilor aflate în cadrul câmpurilor unui formular interactiv sau coordonatelor unei zone senzitive (*image map*). Acest standard conferă interactivitate paginilor Web, documentele HTML putând astfel să-şi modifice în mod dinamic conţinutul şi să permită prelucrări sofisticate de date.

Programele CGI pot fi scrise în orice limbaj, fiind interpretate (cazul Perl, Tcl, scripturi shell Unix) sau compilate (C, C++). Regulile care trebuie respectate la conceperea unui program CGI sunt:

a) programul scrie datele la ieşirea standard (`stdout`);
b) programul generează antete care permit serverului Web să interpreteze corect ieşirea scriptului (folosindu-se specificaţiile protocolului HTTP);
c) datele de intrare se iau de la intrarea standard (`stdin`) sau din variabilele de mediu.

Cele mai multe scripturi CGI sunt concepute pentru a procesa datele introduse în formulare. Un formular se defineşte în XHTML, folosindu-se taguri specifice pentru afişarea conţinutului şi introducerea datelor de către client (vezi specificaţiile pentru XHTML 1.0 ale Consorţiului Web), iar scriptul CGI, executat de server, va prelua conţinutul acelui formular şi-l va prelucra, iar în cele mai multe cazuri va returna un rezultat.

Antetul trimis serverului de către programul CGI va respecta specificaţiile MIME (*Multipurpose Internet Mail Extensions*), conţinând, de exemplu, directiva `Content-type` pentru a indica tipul de document generat. De exemplu, `Content-type: text/html` va indica faptul că rezultatul procesării datelor este un document HTML, `Content-type: image/jpeg` este o imagine în format JPEG, iar `Content-type: text/plain` este text obişnuit.

Tipurile MIME primare sunt sintetizate în următorul tabel:

Tip	Descriere	Exemple
`application`	defineşte aplicaţiile-client	`application/postscript` `application/octet-stream`

`audio`	defineşte formatele audio	`audio/basic`
		`audio/aiff`
		`audio/mid`

`image`	defineşte formatele grafice	`image/gif`
		`image/jpeg`
		`image/png`

`text`	defineşte formatele text	`text/plain`
		`text/html`
		`text/xml`
		`text/css`

➔

`video`	defineşte formatele video	`video/mpeg` `video/quicktime`
`multipart`	utilizat pentru transmisia informaţiilor compuse	`multipart/mixed` `multipart/alternative`
`message`	utilizat pentru transmisia mesajelor de poştă electronică	`message/partial` `message/rfc822`

Tipurile sau subtipurile MIME nestandardizate încă vor fi prefixate de caracterele „`x-`". Astfel, putem întâlni `application/x-cdf` (definiţie de canal de date activ), `image/x-targa` (format grafic Targa) sau `x-world/x-vrml` (lume virtuală modelată în VRML; acest tip MIME este acum înlocuit cu `model/vrml`).

Programului CGI i se vor transmite în variabile de mediu sau de la intrarea standard informaţii referitoare la datele de procesat sau privitoare la clientul care a iniţiat cererea (de exemplu, `SERVER_PORT`, `REQUEST_METHOD`, `SCRIPT_NAME`, `QUERY_STRING`, `REMOTE_HOST` etc.) în funcţie de metoda de transfer utilizată.

10.2.1. Variabilele de mediu

O variabilă de mediu este o variabilă care este iniţializată în afara aplicaţiilor. Aplicaţiile pot accesa aceste variabile. Variabilele de mediu conţin informaţii despre date şi contextul în care se execută aplicaţiile.

De exemplu, variabila de mediu `PATH` conţine o listă cu directoarele unde sunt căutate fişierele executabile atunci când se invocă un program doar prin numele acestuia. Reamintim faptul că variabila de mediu `JAVA_HOME` indică locaţia unde a fost instalat kitul de dezvoltare Java (JDK), iar aplicaţiile pot şti prin intermediul acestei variabile unde sunt fişierele distribuţiei Java.

Oricare program CGI va putea dispune de valorile următoarelor variabile de mediu:

a) Variabile independente de metoda cererii şi setate pentru toate cererile HTTP:
   - `SERVER_SOFTWARE` furnizează numele şi versiunea serverului Web care procesează cererea HTTP şi rulează scriptul CGI;
   - `SERVER_NAME` indică numele *host*-ului, aliasul DNS sau adresa IP a maşinii pe care rulează serverul Web;
   - `GATEWAY_INTERFACE` stochează versiunea specificaţiei CGI folosite (CGI/1.0 ori CGI/1.1 în prezent);

b) Variabile de mediu specifice cererilor care vor fi transmise spre programul CGI:
   - `SERVER_PROTOCOL` indică numele şi versiunea protocolului de transmitere a datelor (*e.g.*, HTTP/1.0);
   - `SERVER_PORT` furnizează portul asociat serverului care va procesa cererea (de obicei, portul 80);
   - `REQUEST_METHOD` specifică metoda prin care va fi formulată cererea: `GET`, `POST`, `PUT` etc.;
   - `SCRIPT_NAME` desemnează calea virtuală până la scriptul care va fi executat;
   - `QUERY_STRING` conţine toate informaţiile de după caracterul `?` din URL-ul care referă scriptul CGI prin metoda `GET`, informaţii care vor trebui decodificate de acel script;

- `REMOTE_HOST` indică *host*-ul care a formulat cererea (calculatorul care a apelat scriptul CGI prin intermediul unei pagini Web);
- `REMOTE_ADDR` furnizează adresa IP a clientului care a formulat cererea, asociată adresei simbolice date de `REMOTE_HOST`;
- `AUTH_TYPE` pune la dispoziţie metoda de autentificare folosită să valideze utilizatorul, dacă serverul suportă autentificarea utilizatorilor şi scriptul este protejat;
- `REMOTE_USER` indică numele utilizatorului, dacă serverul suportă autentificarea utilizatorilor;
- `CONTENT_TYPE` indică tipul conţinutului datelor vehiculate pentru cererile care au ataşate informaţii suplimentare, precum PUT sau POST (de exemplu, pentru un formular transmis prin metoda POST, în mod normal, variabila `CONTENT_TYPE` ar trebui să aibă valoarea `application/x-www-form-urlencoded`);
- `CONTENT_LENGTH` conţine numărul de octeţi ai datelor trimise de client.

Suplimentar, liniile antet (dacă există) recepţionate de la client prin intermediul protocolului HTTP vor fi plasate în variabile de mediu având numele prefixate de `HTTP_`. Ca exemple de astfel de variabile, pot fi menţionate:

- `HTTP_ACCEPT` indică tipurile MIME pe care le acceptă clientul, fiecare tip fiind despărţit de virgulă, aşa cum specifică protocolul HTTP: *tip/subtip*, *tip/subtip*.
- `HTTP_USER_AGENT` furnizează numele şi versiunea navigatorului Web care a formulat cererea, în formatul general: *software/versiune biblioteca/versiune*.

Exemplul 10.2.1. Următorul script Perl va afişa toate variabilele de mediu la care are acces prin intermediul interfeţei CGI:

```
#!/usr/bin/perl
# Afiseaza mediul
print "Content-type: text/plain\n\n";
print "Environment:\n\n";
foreach $key (keys %ENV) {
  printf "%16s = %s\n", $key, $ENV{$key};
}
```

Variabila predefinită `ENV` este un tablou asociativ conţinând perechile (*nume de variabilă de mediu, valoarea variabilei*).

Exemplul 10.2.2. Un script CGI similar, elaborat în *bash* (*Bourne Again Shell*), este:

```
#!/bin/bash
# Afisează mediul
echo "Content-type: text/plain"
echo

set
```

Un posibil rezultat al execuţiei scriptului de mai sus poate fi:

```
HTTP_ACCEPT_LANGUAGE=en
SERVER_SIGNATURE=<ADDRESS>Apache/1.3.12 Server at endirra.ro
                  Port 80</ADDRESS>
SCRIPT_FILENAME=/home/httpd/cgi-bin/env.cgi
HTTP_ACCEPT_CHARSET=iso-8859-1,*,utf-8
HTTP_PRAGMA=no-cache
SERVER_NAME=endirra.ro
HTTP_CONNECTION=Keep-Alive
REMOTE_HOST=thor.infoiasi.ro
REMOTE_ADDR=193.231.30.225
REQUEST_URI=/cgi-bin/env.cgi
HTTP_HOST=endirra
REMOTE_PORT=1041
REQUEST_METHOD=GET
GATEWAY_INTERFACE=CGI/1.1
QUERY_STRING=
SERVER_ADDR=127.0.0.1
SERVER_SOFTWARE=Apache/1.3.12 (Unix)
                 (Red Hat/Linux) PHP/4.0.2 mod_perl/1.21
SERVER_PROTOCOL=HTTP/1.0
SERVER_PORT=80
DOCUMENT_ROOT=/home/httpd/html
HTTP_USER_AGENT=Mozilla/4.75 [en] (X11; U; Linux 2.2.14-5.0 i686)
HTTP_ACCEPT=image/gif, image/x-xbitmap,
             image/jpeg, image/pjpeg, image/png, */*
SCRIPT_NAME=/cgi-bin/env.cgi
SERVER_ADMIN=root@endirra.ro
HTTP_ACCEPT_ENCODING=gzip
```

Aproape pentru orice sistem de operare se poate scrie la linia de comandă `set` pentru a vizualiza valorile de mediu disponibile, precum şi valorile acestora. Vom remarca faptul că rezultatul este diferit faţă de invocarea unei aplicaţii prin intermediul serverului Web. Acesta defineşte şi iniţializează variabilele specifice.

10.2.2. Apelarea programelor CGI din formularele HTML

De cele mai multe ori, un script CGI va fi invocat din cadrul unui formular HTML la apăsarea butonului de trimitere a datelor către server (butonul *submit*).

Exemplul 10.2.3. Un utilizator introduce prin intermediul unui formular două numere întregi, iar programul CGI va genera o pagină Web conţinând suma lor. Formularul XHTML va fi:

```
<form action="http://www.infoiasi.ro/cgi-bin/suma.cgi"
      method="get">
```

```
   <p>Va rugam, introduceti doua numere:
   <input name="nr1" size="5" />
   <input name="nr2" size="5" /> </p>
   <p>
   <input type="submit" value="Afla suma" /></p>
</form>
```

De exemplu, introducând numerele `7` şi `4` şi acţionând butonul etichetat „`Afla suma`", se va primi o pagină Web (document HTML) care va afişa textul „`Suma dintre 7 si 4 este 11`" (aproape întotdeauna, 7 + 4 = 11).

Presupunând că în cele două câmpuri ale formularului (purtând numele `nr1`, respectiv `nr2`) am introdus valorile 7, respectiv 4 şi am apăsat butonul *submit*, navigatorul va trimite, prin intermediul protocolului HTTP, o cerere către serverul Web aflat la adresa dată de URL-ul `http://www.infoiasi.ro/` (adresa este preluată din valoarea atributului `action` din `<form>`). Desigur, în loc de o adresă absolută, putea fi specificată o cale relativă, însemnând faptul că se utilizează serverul pe care se găseşte pagina conţinând formularul.

Atunci când se trimite cererea HTTP, navigatorul construieşte un URL având ca sufix informaţii despre datele introduse în câmpurile formularului, în cazul nostru `http://www.infoiasi.ro/cgi-bin/suma.cgi?nr1=7&nr2=4`, folosindu-se o codificare specială. Pentru fiecare câmp al formularului, se va genera o pereche *nume_câmp = valoare* delimitată de caracterul `&`, iar caracterele speciale (de exemplu, *slash*-ul sau apostroful) vor fi înlocuite de codul lor numeric, în hexazecimal, precedat de caracterul procent (`%`). Spaţiile vor fi substituite de semnul plus (+).

Serverul spre care a fost expediată cererea (în cazul sitului `www.infoiasi.ro`, un server *Apache*) va procesa datele recepţionate conform regulilor proprii. Tipic, configuraţia serverului defineşte unde sunt stocate, în cadrul sistemului de fişiere, directoarele şi fişierele CGI. De cele mai multe ori, serverul Web *Apache* rulează pe maşină sub auspicii de utilizator fictiv (ca `nobody` sau, mai nou, `apache`) din raţiuni de securitate, fişierele sale fiind stocate implicit în directorul `/home/httpd/` sau, mai nou, în `var/www/html`, în cazul unui sistem Unix, iar pentru sistemul Windows în `C:\apache htdocs`. Aici, alături de directorul `html` unde se memorează documentele HTML ale unui sit Web, se află şi directorul `cgi-bin`, unde ar trebui să fie stocate toate fişierele CGI apelate din cadrul paginilor Web. Aşadar, `http://www.infoiasi.ro/cgi-bin suma.cgi` va însemna pentru serverul Web următoarea acţiune: *invocă programu* *`suma.cgi`* *aflat la* *`/home/httpd/cgi-bin`*.

Acţiunea de *invocare* va avea o semantică diferită în funcţie de programul CGI conceput. Pentru un script Perl, serverul va invoca un interpretor Perl (în cazul Apache, un modul special *mod_perl*). Pentru un program executabil (compilat, de pildă, într-unul din limbajele C sau C++, după cum vom vedea mai jos), serverul va lansa programul ca un proces separat.

10.2.3. Procesarea datelor din formularele HTML prin metoda GET

Pentru formularele care utilizează metoda GET (metoda implicită), specificaţiile CGI indică faptul că datele furnizate scriptului vor fi disponibile într-o variabilă de mediu denumită `QUERY_STRING`.

Desigur, în funcţie de limbajul de programare ales pentru conceperea scriptului CGI, datele vor fi accesate via `QUERY_STRING` şi prelucrate. În limbajul C, vom putea folosi funcţia `getenv()` aflată în biblioteca standard `stdlib.h`. Tot ce mai rămâne de făcut este să decodificăm valorile din variabila `QUERY_STRING`.

Exemplul 10.2.4. Scriptul CGI, în mod uzual, va genera date de ieşire care vor fi trimise navigatorului Web de către server, utilizându-se ieşirea standard (*stdout*):

```
#include <stdio.h>
#include <stdlib.h>

#define CONTENT "Content-type: text/html\n\n"

int main(void) {
  char *data;
  long int nr1, nr2;

  /* obligatoriu, trimite tipul de date (în format MIME) */
  printf(CONTENT);
  /* antetul HTML */
  printf("<html>\n<head><title>Rezultat</title></head>\n");
  /* corpul documentului */
  printf("<body bgcolor=\"white\" text=\"black\">\n");
  /* proceseaza datele */
  data = getenv("QUERY_STRING");
  if (data == NULL)
    printf("<p>Eroare la procesarea datelor!</p>\n");
  else
    if(sscanf(data, "nr1=%ld&nr2=%ld", &nr1, &nr2) != 2)
      printf("<p>Eroare! Datele trebuie sa fie numerice!</p>\n");
    else
      printf("<p>Suma dintre %ld si %ld este %ld.</p>\n",
               nr1, nr2, nr1 + nr2);
  /* terminare */
  printf("</body>\n</html>\n");
  return 0;
}
```

Am utilizat specificatorul de format `%ld` pentru că datele numerice sunt de tip `long int`. Din cauza faptului că avem un format fix al datelor recepţionate, am folosit `sscanf()`, dar pentru decodificări mai complexe trebuie să scriem o funcţie specială (vezi mai jos).

Executabilul va trebui plasat după compilare (în cele mai multe situaţii) în directorul `cgi-bin` al serverului Web. Pentru a putea fi accesat, este necesar să posede drepturi de execuţie pentru serverul Web (de obicei, pentru utilizatorul special `nobody` sau `apache` în Unix).

După instalarea scriptului, orice utilizator din internet (dacă nu sunt impuse restricţii de către administratorul Web) îl va putea accesa şi executa prin intermediul unui formular HTML sau explicit. Astfel, scriptul va trebui să fie capabil să proceseze orice tip de dată şi să semnaleze posibilele erori.

Însă metoda GET are câteva neajunsuri: poate trimite un număr limitat de caractere (octeţi) prin intermediul variabilelor (de regulă 255 octeţi), iar câmpurile de parolă vor fi vizibile în URL, ceea ce oferă o securitate redusă. Totuşi, această metodă este frecvent utilizată, mai ales de motoarele de căutare.

10.2.4. Procesarea unui formular prin metoda POST

Un program CGI are acces la întreg mediul sistemului de operare pe care rulează. Metoda POST este utilă în situaţiile în care avem de trimis scriptului CGI spre prelucrare un volum mai mare de date (de exemplu, conţinutul unei scrisori introdus prin intermediul unui `<textarea>` ori al unui fişier, prin acţiunea de ***upload***) sau informaţii confidenţiale (*e.g.*, parole) care nu trebuie să apară în componenţa URL-ului transmis serverului Web.

Pentru formularele utilizând metoda POST, datele trimise scriptului vor putea fi accesate de la intrarea standard, iar lungimea, în octeţi, a datelor trimise va fi disponibilă în variabila de mediu `CONTENT_LENGTH`. Vor trebui citite cel mult atâtea caractere câte indică variabila de mediu `CONTENT_LENGTH`.

Exemplul 10.2.5. Formularul următor permite introducerea unui titlu de proiect care va fi adăugat la sfârşitul unui fişier aflat pe server:

```
<form action="http://www.infoiasi.ro/cgi-bin/addproj.cgi"
      method="POST">
  <p>Introduceti titlul proiectului:
  <input name="data" size="50" />
  <input type="submit" value="Trimite" /></p>
</form>
```

Programul C care acceptă datele şi le adaugă la fişierul `proj.txt` este următorul:

```
#include <stdio.h>
#include <stdlib.h>

#define MAXLEN 80
```

```
#define EXTRA 5
/* 4 pentru numele campului "data", 1 pentru "=" */
#define MAXINPUT MAXLEN+EXTRA+2
/* 1 pentru sfarsit de linie, 1 pentru NULL */

/* numele fisierului (cale relativa) */
#define DATAFILE "proj.txt"

/* functie de decodificare a datelor transmise programului CGI */
void unencode(char *src, char *last, char *dest)
{
 for (; src != last; src++, dest++)
   if (*src == '+') /* inlocuieste "+" cu spatiu */
     *dest = ' ';
   else
     if (*src == '%')
       {            /* inlocuieste hexa cu caracterul */
         int code;
         if (sscanf(src + 1, "%2x", &code) != 1)
           code = '?';
         *dest = code;
         src +=2;
       }
     else             /* copiaza celelalte caractere... */
     *dest = *src;
 *dest = '\n';
 *++dest = '\0';
}

/* functia principala */
int main(void)
{
  char *lenstr;
  char input[MAXINPUT], data[MAXINPUT];
  long len;

 /* scrie Content-type, urmat de setul de caractere (optional),
   in acest caz ISO-8859-2 (folosim diacritice) */
  printf("%s%c%c\n",
    "Content-type: text/html;charset=iso-8859-2", 13, 10);

  /* scrie antetul documentului HTML generat */
  printf("<html>\n<head><title>R&#259;spuns</title></head>\n");

  /* ia numarul de caractere transmise */
  lenstr = getenv("CONTENT_LENGTH");

  /* numarul de caractere este corect? */
  if (lenstr == NULL ||
      sscanf(lenstr, "%ld", &len) != 1 ||
      len > MAXLEN)
    printf("<p>Eroare - formular incorect?\n");
  else
  {
    FILE *f;
    /* citeste de la intrarea standard datele */
    fgets(input, len + 1, stdin);
    /* decodifica linia */
    unencode(input + EXTRA, input + len, data);
    /* adauga în fisier */
    f = fopen(DATAFILE, "a");
    if (f == NULL)
      printf("<p>Eroare la ad&#259;ugarea &icirc;n" +
        "fi&#351;ier...");
    else
      fputs(data, f);
    fclose(f);
    printf("<p>Mul&#355;umim! Linia <b>%s</b> a fost ad&#259;
    ugat&#259; &icirc;n fi&#351;ier</p>", data);
  }
  /* sfarsitul documentului */
  printf("</body>\n</html>\n");
  return 0;
}
```

Propunem cititorului să conceapă, în limbajul C, scriptul CGI care să realizeze vizualizarea informaţiilor despre proiecte, prin consultarea conţinutului fişierului `proj.txt`.

10.3. Servleturi

Un *servlet* este o componentă software pe partea de server, scrisă în Java, şi care extinde dinamic funcţionalitatea unui server (de obicei, un server HTTP). Reamintim că un applet este o componentă software pe partea clientului care rulează în cadrul unui navigator Web. Spre deosebire de appleturi, servleturile nu afişează o interfaţă grafică către utilizator, ci returnează doar rezultate către client (de obicei, sub forma unui document HTML).

De fapt, servleturile sunt clase Java ce se conformează unei interfeţe specifice care poate fi apelată de către server. Funcţionalitatea furnizată de un servlet nu priveşte numai serverele Web, ci se poate extinde la orice server care suportă Java şi Servlet API (cum ar fi: FTP, Telnet, mail, servere de ştiri). Servlet API este o specificare dezvoltată de Sun Microsystems (şi alţii) care defineşte clasele şi interfeţele necesare pentru crearea şi execuţia servleturilor.

Servleturile furnizează un cadru pentru crearea de aplicaţii care implementează paradigma cerere/răspuns. De exemplu, când un navigator trimite o cerere către server, serverul o trimite mai departe unui servlet. Servletul procesează cererea (eventual accesând o bază de date) şi construieşte un răspuns convenabil (de obicei în HTML), care este returnat clientului. Servleturile funcţionează asemănător CGI-urilor.

10.3.1. Servlet API

Servlet API (*Application Programming Interface*) este o extensie Java standard a platformei Java 2, ediţia standard (J2SE – *Java 2 Standard Edition*). Toate extensiile Java sunt conţinute în pachetul `javax`. Pachetele pentru servleturi sunt `javax.servlet` şi `javax.servlet.http`. Servlet API este disponibil gratuit, ca o bibliotecă de sine stătătoare Java (*http://java.sun.com/products/servlet*) sau ca parte componentă a proiectului Apache Tomcat (*http://tomcat.apache.org/*).

Spre deosebire de ediţia standard, Servlet API este un membru obligatoriu al platformei *Java 2 Enterprise Edition* (J2EE).

10.3.2. Funcţionalitatea servleturilor

Ca şi CGI-urile, servleturile permit interacţiunea în ambele sensuri între client şi server. Iată o parte dintre posibilităţile furnizate de servleturi:

- construiesc dinamic şi returnează un document HTML pe baza cererii clientului;
- procesează datele completate de utilizatori printr-un formular HTML şi returnează un răspuns;
- furnizează suport pentru autentificarea utilizatorului şi alte mecanisme de securitate;
- interacţionează cu resursele serverului, cum ar fi baze de date, fişiere cu informaţii utile pentru clienţi;
- procesează intrările de la mai mulţi clienţi pentru aplicaţii, cum ar fi jocurile în reţea;
- permite serverului să comunice cu un applet-client printr-un protocol specific şi păstrează conexiunea în timpul conversaţiei;
- ataşează automat elemente de design pentru pagini Web, cum ar fi antete sau note de subsol, pentru toate paginile returnate de server;
- redirectează cereri de la un server la altul în scop de echilibrare a încărcării (engl. *load-balancing*);
- partiţionează un serviciu logic între servere sau între servleturi pentru a procesa eficient o problemă.

Există şi alte soluţii software care extind funcţionalitatea unui server HTTP:

1. FastCGI, dezvoltat iniţial de firma *Open Market*, îmbunătăţeşte CGI-ul standard, însă nu se poate compara cu performanţa şi eficienţa unui servlet Java. De exemplu, spre deosebire de servleturi, FastCGI necesită cel puţin un proces pentru fiecare program şi fiecare proces poate servi doar o cerere la un moment dat;
2. NSAPI/ISAPI (*Netscape Server API/Microsoft Internet Server API*) sunt interfeţe pentru extinderea funcţionalităţii unui server HTTP. Acestea furnizează mai multă performanţă decât servleturile, însă sunt legate de servere HTTP şi platforme particulare. De exemplu, ISAPI este suportat de *Microsoft Internet Information Server* (IIS) care rulează pe platforma *Windows NT/2000/XP/ .NET*. Similar, aplicaţiile NSAPI sunt compatibile doar cu câteva servere Web din *Netscape/iPlanet*. Chiar dacă serverele NSAPI sunt disponibile sub Windows şi UNIX, din moment ce programele sunt scrise în C, aceste aplicaţii trebuie recompilate pentru fiecare nouă platformă;
3. alte soluţii sunt oferite de PHP, Zope sau Lotus Notes/Domino.

10.3.3. Execuţia servleturilor

Există multe posibilităţi de a executa servleturi. Anumite servere Web, cum ar fi *iPlanet Web Server* şi *W3C Jigsaw*, oferă suport nativ pentru servleturi. Alte servere HTTP, cum ar fi *Apache Web Server* şi *Microsoft Internet Information Server*, oferă suport prin produsele care se pot instala:

- *Tomcat* de la *Apache*;
- *JRun* de la *Allaire*;
- *ServletExec* de la *New Atlanta Communications*;
- *Resin* de la *Caucho Technology*.

În continuare, vom prezenta cel mai simplu server, care este şi gratuit: Apache Tomcat. Acesta este o implementare de referinţă oficială pentru servleturile Java şi pentru specificaţiile JSP (engl. *Java Server Pages*).

Se instalează local, dezarhivând fişierele copiate sau utilizând programul de instalare. Urmează pornirea serverului Tomcat prin comanda `startup` sau prin pornirea serviciului creat la instalare. Se poate verifica dacă instalarea şi pornirea serverului Tomcat s-au făcut cu succes dacă atunci când se solicită unui navigator Web adresa *http://localhost:8080/*, se obţine pagina de lucru a serverului Tomcat. Se poate modifica numărul portului (8080) editând şi schimbând porţiunea de cod corespunzătoare din fişierul `tomcat/conf/server.xml`.

Lucrurile se petrec similar pe platforma Unix/Linux.

Toate servleturile specifice standardului Servlet API utilizează conceptul de aplicaţie Web. O *aplicaţie Web* este o ierarhie de directoare şi fişiere care formează împreună o aplicaţie. De obicei, acestea sunt distribuite într-un fişier de aplicaţii Web arhivate (în general, cu extensia `.war`). Toate aplicaţiile Web utilizează aceeaşi structură de directoare referitoare la serverul utilizat. Însă locul unde sunt instalate aplicaţiile Web poate să difere de la un server la altul. De exemplu, Tomcat păstrează toate aplicaţiile Web în directorul `webapps`, iar JRun păstrează aplicaţiile Web pentru serverul implicit în `servers/default`.

În subdirectoarele directorului `webapps`, serverul Tomcat va căuta fişiere HTML, JSP şi imaginile asociate unei aplicaţii Web. În general, fiecărei aplicaţii Web i se atribuie un „drum de context" unic de către administratorul serverului Web. Toate cererile către acest drum de context vor fi direcţionate către aplicaţia Web corespunzătoare.

Dacă aplicaţiei Web i se atribuie drumul de context `/examples`, atunci URL-ul *http://localhost:8080/examples/index.html* va afişa fişierul `index.html` aflat în directorul `webapps/examples`.

Drumul de context pentru fiecare aplicaţie Web este specificat în fişierul de configurare `server.xml` din directorul `conf` al serverului Tomcat.

În plus, o aplicaţie Web poate fi definită prin specificarea unui drum de context vid. De exemplu, Tomcat instalează aplicaţia `ROOT` ca aplicaţie implicită prin asignarea contextului vid. De exemplu, URL-ul *http://localhost:8080/index.jsp* va returna fişierul `index.jsp` din directorul `webapps/ROOT`.

Apelarea servleturilor utilizând serverul Tomcat se realizează după următorul format pentru URL:

```
http://<server>:<port>/<drum_context>/servlet/<nume_servlet>
    [/<informatii_drum>][?<sir_interogare>]
```

Identificatorii scrişi între paranteze unghiulare (<, >) semnifică faptul că aceştia trebuie înlocuiţi. `<drum_context>` reprezintă locul unde se memorează aplicaţia Web (acesta trebuie să fie unic). Cuvântul `servlet` indică serverului Tomcat că este vorba de un servlet şi nu o pagină HTML sau JSP; `<nume_servlet>` reprezintă numele clasei servletului sau numele servletului (alias); `<informatii_drum>` şi `<sir_interogare>` sunt componente suplimentare pentru un URL. `<informatii_ drum>` se referă la informaţii suplimentare. Acestea pot fi fi obţinute aplicând metoda `getPathInfo()` a obiectului `HttpServletRequest`. De exemplu, dacă se apelează un servlet numit `ObtineCaleServlet` prin URL-ul:

```
http://localhost:8080/servlet/ObtineCaleServlet/html/public?
name=value
```

atunci valoarea returnată de metoda `getPathInfo()` este `/html/public`.

`<sir_interogare>` permite trimiterea de informaţii suplimentare unui servlet utilizând formatul `nume=valoare`. Şirurile `nume=valoare` din interogare (engl. *query*) pot fi citite apelând pentru un obiect `HttpServletRequest` metodele `get ParameterNames()`, `getParameter()` şi/sau `getParameterValue()`.

10.3.4. Structura de bază a unui servlet

Cel mai uşor mod de definire a servleturilor este prin extinderea uneia dintre cele două clase de bază referitoare la servleturi: `GenericServlet` şi `HttpServlet`. De fapt, nu este obligatorie moştenirea acestor clase, ci este suficientă implementarea interfeţei `Servlet`.

Toate servleturile suprascriu cel puţin o metodă necesară funcţionalităţii lor. Metoda care este automat apelată ca răspuns la cererea fiecărui client se numeşte `service()`. Această metodă poate fi suprascrisă pentru a furniza o funcţionalitate implicită. Cu toate acestea, servleturile care extind `HttpServlet` pot să nu suprascrie metoda `service()`. În acest caz, implementarea implicită a acestei metode va apela automat altă metodă pentru a răspunde cererii clientului.

Mai există încă două metode implementate de majoritatea servleturilor: `init()` şi `destroy()`. Metoda `init()` se apelează o singură dată, când este încărcat servletul (similar unui constructor). Metoda `destroy()` este apelată când servletul este descărcat, eliberând resursele ocupate de servlet.

Un schelet pentru servleturi poate fi (fără a preciza parametrii şi excepţiile):

```
public class ScheletServlet extends HttpServlet  {
  public void init() {
       // codul de initializare
  }
  public void service() {
       // partea efectiva de lucru a servletului
  }
  public void destroy() {
       // eliberarea resurselor
  }
} // sfarsitul clasei ScheletServlet
```

Dacă servletul extinde `HttpServlet`, dezvoltatorul servletului va implementa altă metodă care va fi automat apelată de metoda moştenită `service()`. Metoda automat apelată de `service()` depinde de tipul cererii HTTP (de exemplu, `doGet()` este apelată pentru cererile prin metoda `GET` şi `doPost()` este apelată pentru cererile `POST`).

10.3.5. Ciclul de viaţă al unui servlet

În principiu, procesul apelării de către server a unui servlet se poate împărţi în opt paşi:

1. Serverul încarcă servletul când acesta este cerut de client sau la startarea serverului, dacă aşa impune configuraţia. Servletul poate fi încărcat local sau dintr-o locaţie de la distanţă folosind o facilitate de încărcare a claselor Java. De exemplu, putem avea:

```
Class c = Class.forName("surseFlux.ServletUnu");
```

2. Serverul creează o instanţă a clasei servletului pentru deservirea tuturor cererilor. Folosind fire de execuţie multiple, cererile concurente pot fi deservite de o singură instanţă a servletului. De exemplu, putem avea:

```
Servlet s = (Servlet) c.newInstance();
```

3. Serverul apelează metoda `init()` a servletului care garantează că termină execuția înaintea procesării primei cereri a servletului. Dacă serverul creează mai multe instanțe ale servletului, metoda `init()` se apelează de fiecare dată pentru fiecare instanță.
4. În momentul primirii unei cereri pentru servlet, serverul construiește un obiect `ServletRequest` sau `HttpServletRequest` din datele incluse în cererea clientului. De asemenea, acesta construiește un obiect `ServletResponse` sau `HttpServletResponse` care furnizează metode pentru returnarea răspunsului. Tipul parametrului depinde dacă servletul extinde `GenericServlet`, respectiv `HttpServlet`.
5. Serverul apelează metoda `service()` (care pentru servleturi HTTP apelează, la rândul ei, o metodă specifică, cum ar fi `doGet()` sau `doPost()`), trimițând ca parametri obiectele construite la pasul 4. Când sosesc cereri concurente, metodele `service()` pot rula simultan în fire de execuție separate (cu excepția situației când servletul implementează interfața `SingleThreadModel`).
6. Metoda `service()` procesează cererea clientului prin evaluarea obiectului `ServletRequest` sau `HttpServletRequest`. Apoi acesta răspunde, utilizând obiectul `ServletResponse` sau `HttpServletResponse`.
7. Dacă serverul primește încă o cerere pentru acest servlet, procesul începe din nou la pasul 5.
8. De fiecare dată când containerul servletului determină că un servlet trebuie descărcat (din motive de corecție sau dacă acesta a „căzut"), serverul apelează metoda `destroy()` după terminarea firelor de execuție ale metodei `service()` sau după terminarea limitei de timp definite de server. Servletul poate porni dealocatorul de memorie (*garbage collector*).

Vom prezenta mai întâi un servlet generic. Spre deosebire de servleturile HTTP, servleturile generice nu sunt specifice unui protocol anume. Servleturile generice (care extind clasa `GenericServlet`) sunt utilizate, de obicei, în aplicații de rețea care implică un protocol obișnuit sau orice alt protocol diferit de HTTP, cum ar fi FTP, SMTP sau POP3. Exemplul de mai jos conține un servlet generic care întoarce timpul curent de la server.

Exemplul 10.3.1. Servlet generic:

```
import javax.servlet.*;

/** Acest servlet returneaza timpul curent  de la server */
public class ServletDeTimp extends GenericServlet {

  /** Serverul apeleaza metoda service() pentru a raspunde
  la cererile adresate servletului */
  public void service(ServletRequest cerere,
      ServletResponse raspuns)
          throws ServletException, java.io.IOException {
```

```
      // declaram o referinta catre fluxul de iesire al clientului
        java.io.PrintWriter iesireClient = raspuns.getWriter();

        // tiparim timpul curent si data catre fluxul de iesire
        iesireClient.println(new java.util.Date());
        iesireClient.close();
    }
}
```

Fişierul `ServletDeTimp.class` va fi stocat în directorul `webapps/examples/WEB-INF/classes`, iar în navigatorul Web se va insera adresa `http://localhost:8080/examples/servlet/ServletDeTimp`. Servletul va afişa în fereastra navigatorului Web timpul în care a fost executat servletul.

10.3.6. Clasa `HttpServlet`

Servleturile HTTP (cele care extind clasa `HttpServlet`) sunt utilizate pentru construirea de aplicaţii Web, care, de obicei, întorc documente Web (XHTML, WML, XML etc.) ca răspuns la cererile navigatorului.

Clasa `HttpServlet` extinde clasa `GenericServlet`, deci moşteneşte toate funcţionalităţile acesteia. În plus, `HttpServlet` adaugă funcţionalitatea specifică protocolului HTTP şi furnizează un cadru în care putem construi aplicaţii HTTP.

`HttpServlet` este o clasă abstractă prezentă în pachetul `javax.servlet.http`. Pentru că este abstractă, ea nu poate fi instanţiată. Când se construieşte un servlet HTTP, trebuie extinsă clasa `HttpServlet`. Un servlet HTTP funcţional trebuie să implementeze una dintre metodele: `service()`, `doGet()`, `doHead()`, `doPost()`, `doPut()`, `doDelete()` corespunzătoare metodelor protocolului HTTP.

Metodele `init()`, `destroy()` şi `getServletInfo()` aparţin clasei `GenericServlet`, din care se derivează clasa `HttpServlet`.

Metoda `service()` are prototipul:

```
public void service(HttpServletRequest cerere,
   HttpServlet Response raspuns)
      throws ServletException, java.io.IOException;
```

Primul argument al metodei `service()` este un obiect care implementează interfaţa `HttpServletRequest` şi este reprezentarea cererii clientului. Toate datele relevante din cerere (cum ar fi antetele HTTP) sunt încapsulate într-un obiect `HttpServlet Request`. Al doilea parametru este un obiect care implementează interfaţa `Http ServletResponse` şi este reprezentarea răspunsului serverului către client.

Când se utilizează servleturi HTTP, este recomandată utilizarea metodelor de cerere specifice HTTP, cum ar fi `doGet()` sau `doPost()`, în locul suprascrierii metodei `service()`.

În general, metoda `service()` nu se suprascrie. Asta, deoarece clasa `HttpServlet` defineşte metode care se apelează pentru a răspunde diferitelor tipuri specifice de cereri HTTP. Dacă metoda `service()` nu este suprascrisă, atunci implementarea implicită este să se apeleze o metodă de tipul `doXxx()` care corespunde tipului cererii realizate. Dacă metoda `service()` este suprascrisă şi sunt implementate anumite metode `doXxx()`, este sarcina dezvoltatorului de servlet să evalueze cererea HTTP şi să apeleze metoda `doXxx()` corespunzătoare. Un container de servleturi nu apelează direct metodele `doXxx()`, ci metoda `service()`.

Dacă se încearcă apelarea unei metode `doXxx()`, care însă nu este suprascrisă, atunci se primeşte la client un mesaj de eroare: `HTTP 400 "Bad Request"`.

Exemplul 10.3.2. Supradefinirea metodei `service()`:

```
import javax.servlet.http.*;
import javax.servlet.ServletException;
import java.io.*;

public class Service extends HttpServlet
{
  public void service(HttpServletRequest cerere,
    HttpServletResponse raspuns)
    throws ServletException, IOException {

    // setam tipul MIME pentru antetul HTTP
    raspuns.setContentType("text/html");

    // obtinem o referinta la fluxul de iesire
    PrintWriter iesire = raspuns.getWriter();

    // determinam tipul cererii
    if (cerere.getMethod().equalsIgnoreCase("GET")) {
      iesire.println(
        "O cerere GET rezolvata prin metoda service()");
    }
    else { // nu este o cerere GET
    // trimitem un mesaj de eroare din moment ce doar cererile
    // GET sunt suportate
      raspuns.sendError(HttpServletResponse.SC_BAD_REQUEST);
    }
    iesire.close(); // inchidem fluxul de iesire
  }
}
```

Metoda `doGet()` are prototipul:

```
protected void doGet(HttpServletRequest cerere,
   HttpServletResponse raspuns)
     throws ServletException, java.io.IOException;
```

Metoda `doGet()` este apelată de implementarea implicită a metodei `service()` ca răspuns la o cerere HTTP de tip `GET`. Ca şi alte metode de tip `doXxx()`, metoda `doGet()` are ca parametri o reprezentare a cererii clientului şi un răspuns al serverului. Citirea obiectului cerere şi formarea (trimiterea) obiectului răspuns este funcţia principală a unui servlet.

Exemplul 10.3.3. Procesarea unei cereri prin metoda `GET`:

```
import javax.servlet.http.*;
import javax.servlet.ServletException;
import java.io.*;

public class GetUnu extends HttpServlet
{
  /** Metoda returneaza un mesaj scurt catre client */
  protected void doGet(HttpServletRequest cerere,
    HttpServletResponse raspuns)
      throws ServletException, IOException {

    // setam tipul MIME pentru antetul HTTP
    raspuns.setContentType("text/html");

    // obtinem o referinta la fluxul de iesire
    PrintWriter iesire = raspuns.getWriter();

    // trimitem un raspuns
    iesire.println(
      "O cerere GET rezolvata prin metoda doGet()");

    // inchidem fluxul de iesire
    iesire.close();
  }
}
```

Metoda `doHead()` a fost eliminată din Servlet API 2.2 şi reintrodusă în Servlet API 2.3 pentru a rezolva cererile HTTP de tip `HEAD`. Dacă metoda `doHead()` nu este suprascrisă de servlet, implementarea implicită va apela metoda `doGet()` pentru a genera câmpuri antet corespunzătoare. Apoi, se va returna clientului un răspuns ce conţine doar antetul HTTP (nu şi conţinutul documentului generat).

Continuăm cu prezentarea metodei `doPost()` care are prototipul:

```
protected void doPost(HttpServletRequest cerere,
    HttpServletResponse raspuns)
      throws ServletException, java.io.IOException;
```

Metoda `doPost()` este apelată de fiecare dată când apare o cerere HTTP de tip POST primită de servlet. Această metodă este, de obicei, utilizată pentru procesarea informaţiilor colectate dintr-un formular Web. Informaţia introdusă de utilizator într-un formular HTML este încapsulată într-un obiect `HttpServletRequest` şi trimisă metodei `doPost()`. Dacă nu este găsită o implementare a metodei `doPost()`, atunci se trimite către client un mesaj de eroare: `HTTP 400 "Bad Request"`.

Pentru a remarca funcţionalitatea unui servlet, vom prezenta două exemple de utilizare a metodelor `doGet()` şi `doPost()`.

Exemplul 10.3.4. Presupunem că am scris în fişierul `formular.html` următorul cod XHTML:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>Un exemplu de adunare</title>
  <!-- Specificarea setului de caractere pentru diacritice -->
  <meta http-equiv="Content-Type"
     content="text/html; charset=iso-8859-2" />
</head>

<body>
  <h1>Suma a dou&#259; numere reale</h1>
  <p>Da&#355;i cele doua numere! :)</p>
  <form method="GET"
     action="http://localhost:8080/examples/servlet/SumaGet">
  <p>Num&#259;rul 1: <input type="text" name="nr1" /></p>
  <p>Num&#259;rul 2: <input type="text" name="nr2" /></p>
  <p><input type="submit" value="Trimite" />
    &nbsp;<input type="reset" value="&#350;terge" /> </p>
</form>
</body>
</html>
```

Se observă că acest fişier utilizează metoda `GET`. Plasăm în directorul `webapps\examples\WEB-INF\classes` fişierul `SumaGet.class` corespunzător codului-sursă Java:

```
import javax.servlet.http.*;
import javax.servlet.ServletException;
import java.io.*;

public class SumaGet extends HttpServlet {
  // procesarea formularului XHTML
  protected void doGet(HttpServletRequest cerere, HttpServlet
    Response raspuns) throws ServletException, IOException {
    String numarUnu = cerere.getParameter("nr1");
    String numarDoi = cerere.getParameter("nr2");
    double nr1 = 0.0, nr2 = 0.0;
    Double d1Temp = Double.valueOf(numarUnu);
    nr1 = d1Temp.doubleValue();
    d1Temp = Double.valueOf(numarDoi);
    nr2 = d1Temp.doubleValue();
    double suma = nr1 + nr2;

    PrintWriter iesire = raspuns.getWriter();
    iesire.println("<html><head>");
    iesire.println("<title>Un exemplu de adunare</title>");
    iesire.println("</head>");
    iesire.println("<body>");
    iesire.println("<h1>Suma celor doua numele</h1>");
    iesire.println("<p>Suma celor doua numere reale este " -
       suma + "</p>");
    iesire.println("</body></html>");
    iesire.close(); // inchidem fluxul de iesire
  }
}
```

Deschizând documentul XHTML de mai sus într-un navigator Web, după completarea câmpurilor şi apăsarea butonului *Submit*, se va apela servletul de mai sus apelând metoda `doGet()`, care va trimite către client o pagină Web ce va afişa suma celor două numere introduse în formular.

În continuare, vom prezenta o variantă mai elegantă de a realiza suma a două numere. Astfel, vom vedea o funcţionalitate mai mare a servletului, utilizând ambele metode – `doGet()` şi `doPost()`.

Exemplul 10.3.5. Utilizarea celor două metode de prelucrare a formularelor: `GET` şi `POST`.

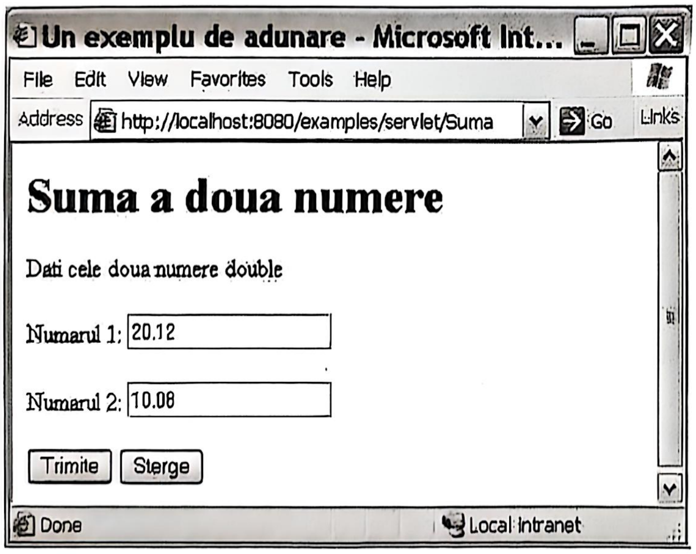
```
import javax.servlet.http.*;
import javax.servlet.ServletException;
import java.io.*;

public class Suma extends HttpServlet {

  /** pentru metoda GET
   returnam un formular HTML pentru citirea a doua numere *
  protected void doGet(HttpServletRequest cerere,
    HttpServlet  Response raspuns)
      throws ServletException, IOException {

    // setam tipul MIME pentru antetul HTTP
    raspuns.setContentType("text/html");
   // scriem documentul Web cu formularul
   PrintWriter iesire = raspuns.getWriter();
   iesire.println("<html><head>");
   iesire.println("<title>Un exemplu de adunare</title>");
   iesire.println("</head>");
   iesire.println("<body>");
   iesire.println("<h1>Suma a doua numere</h1>");
   iesire.println("<p>Dati cele doua numere double</p>");
   iesire.println("<form method=\"POST\" action=" +
      "\"http://localhost:8080/examples/servlet/Suma\">");
   iesire.println(
      "<p>Numarul 1: <input type=\"text\" name=\"nr1\"></p>");
   iesire.println("<p>Numarul 2: <input type=\"text\" " +
      "name=\"nr2\"></p>");
   iesire.println(
      "<p><input type=\"submit\" value=\"Trimite\">&nbsp;" +
      "<input type=\"reset\" value=\"Sterge\"></p>");
   iesire.println("</form></body></html>");
   iesire.close(); // inchidem fluxul de iesire
}

// pentru metoda POST
// procesam datele primite din formularul HTML si returnam
// un document HTML care afiseaza suma celor doua numere
protected void doPost(HttpServletRequest cerere, HttpServlet
   Response raspuns)throws IOException {
   // setam tipul MIME pentru antetul HTTP
   raspuns.setContentType("text/html");

   // preluam parametrii
   String numarUnu = cerere.getParameter("nr1");
   String numarDoi = cerere.getParameter("nr2");
   double nr1 = 0.0, nr2 = 0.0;
   Double d1Temp = Double.valueOf(numarUnu);
   nr1 = d1Temp.doubleValue();
   d1Temp = Double.valueOf(numarDoi);
   nr2 = d1Temp.doubleValue();

   // efectuam suma
   double suma = nr1 + nr2;

   // scriem rezultatul
   PrintWriter iesire = raspuns.getWriter();
   iesire.println("<html><head>");
   iesire.println("<title>Un exemplu de adunare</title>");
```

```
        iesire.println("</head>");
        iesire.println("<body>");
        iesire.println("<h1>Suma celor doua double</h1>");
        iesire.println("<p>Suma celor doua numere double este " +
          suma + "</p>");
        iesire.println("</body></html>");
        iesire.close(); // inchidem fluxul de iesire
    }
} // sfarsitul definitiei clasei Suma
```

În exemplul acesta, metoda doGet() este utilizată pentru crearea unui formular Web. La apăsarea butonului pentru *Submit*, datele formularului sunt trimise către servlet şi procesate de metoda doPost(). Aceasta construieşte un document HTML care conţine suma celor două numere trimise şi apoi le returnează clientului.

De altfel, când se doreşte citirea de informaţii de la client, se procedează, de obicei, în acest fel: utilizând metoda doGet(), se creează un formular HTML şi apoi se prelucrează datele primite returnându-se o pagină Web cu ajutorul metodei doPost().

Execuţia servletului poate fi împărţită în două etape, trimiterea datelor şi primirea rezultatului, astfel:


Un exemplu de adunare - Microsoft Int...
File Edit View Favorites Tools Help
Address
Go
Links
Trimite
Sterge
Done
Local intranet

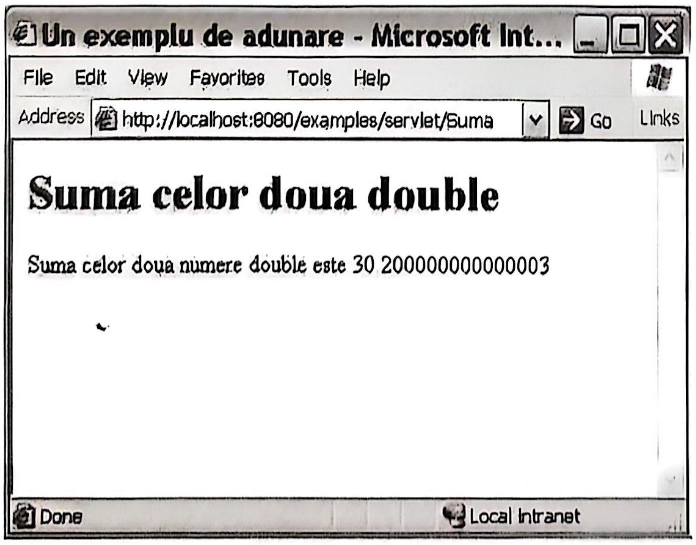
http://localhost:8080/examples/servlet/Suma

Suma a doua numere

Dati cele doua numere double

Numarul 1: 20.12

Numarul 2: 10.08


Un exemplu de adunare - Microsoft Int...
File Edit View Favorites Tools Help
Address
Go
Links
Done
Local intranet

http://localhost:8080/examples/servlet/Suma

Suma celor doua double

Suma celor doua numere double este 30.200000000000003

10.3.7. Redirectarea cererii

Redirectarea unei cereri constă în trimiterea unui URL ca răspuns, navigatorul Web facând apel la resursa respectivă pentru a afişa un rezultat. Acest lucru se poate realiza prin apelul metodei `sendRedirect()` din interfaţa `HttpServerResponse`. Această metodă are un parametru de tip `String` ce conţine URL-ul la care se realizează redirectarea.

Exemplul 10.3.6. Redirectarea cererii la adresa *http://localhost:8080/examples/servlet/ServletDeTimp*:

```
import javax.servlet.*;
import javax.servlet.http.*;

public class Redirectare extends HttpServlet {
  /** Tratarea cererilor prin metoda GET */
  public void doGet(HttpServletRequest cerere,
    HttpServletResponse raspuns)
      throws ServletException, java.io.IOException {
    /** redirectarea cererii */
    raspuns.sendRedirect(
      "http://localhost:8080/examples/servlet/ServletDeTimp");
  }
}
```

Dacă în navigatorul Web se va scrie adresa *http://localhost:8080/examples/servlet/* Redirectare, atunci se va face apel la servletul *http://localhost:8080/examples/servlet/* ServerDeTimp (adresă care va apărea şi în bara de adrese a navigatorului).

10.3.8. Returnarea unei erori

Servleturile pot întoarce coduri de eroare conform protocolului HTTP. Fiecare cerere HTTP primeşte ca răspuns un cod de stare care va indica tipul de răspuns primit (dacă este vorba de o informare, de succes sau eşec etc.). Pentru mai multe detalii despre codurile HTTP se poate consulta [PSW05], sau specificaţiile protocolului HTTP.

Pentru a întoarce un astfel de cod de stare, interfaţa `HttpServletResponse` pune la dispoziţie metoda `sendError()`. Aceasta are două forme:

- `void sendError(int cod)`
- `void sendError(int cod, String mesaj)`

unde `cod` este codul de stare, iar `mesaj` este mesajul care însoţeşte respectivul cod.

Exempul 10.3.7. Trimiterea unui cod de stare:

```
import javax.servlet.*;
import javax.servlet.http.*;
```

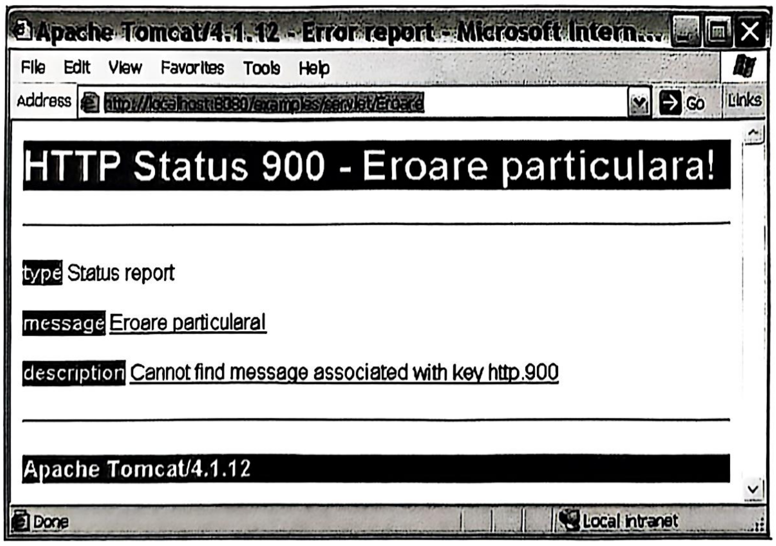
```
public class Eroare extends HttpServlet {
  /** Procesarea cererilor prin metoda GET */
  protected void doGet(HttpServletRequest cerere,
    HttpServletResponse raspuns)
      throws ServletException, java.io.IOException {
    /** trimiterea codului de stare 404 */
    raspuns.sendError(404);
  }
}
```

Servletul de mai sus returnează codul de stare 404. Acesta corespunde situaţiilor când URL-ul indicat este invalid (nu se află nici o resursă la adresa indicată).

Se pot utiliza şi coduri neutilizate în specificaţiile protocolului HTTP. Spre exemplu, dacă înlocuim conţinutul metodei `doGet()` din exemplul anterior cu următoarea linie:

```
raspuns.sendError(900,"Eroare particulara!");
```

atunci se va obţine un cod care nu va putea fi interpretat de navigatoarele Web şi va fi afişat asemenea unui mesaj de eroare. Codul `900` va fi însoţit şi de mesajul specificat, după cum se poate vedea şi în figura de mai jos:


Apache Tomcat/4.1.12 - Error report - Microsoft Intern...
File
Edit
View
Favorites
Tools
Help
Address
http://localhost:8080/examples/servlet/Eroare
Go
Links
Done
Local intranet

HTTP Status 900 - Eroare particulara!

type Status report

message Eroare particulara!

description Cannot find message associated with key http.900

Apache Tomcat/4.1.12

10.3.9. Exemplu de aplicaţie Web

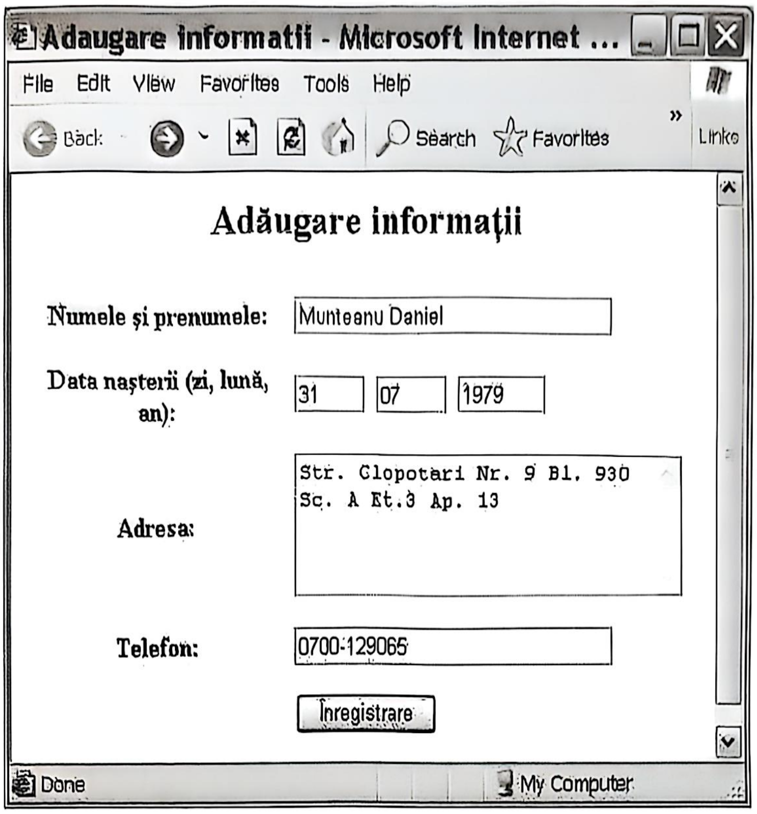
În această secţiune vă oferim un exemplu de aplicaţie Web simplă, implementată cu ajutorul servleturilor. Aplicaţia permite stocarea informaţiilor despre persoane (nume, data naşterii, adresă şi număr de telefon) prin intermediul unui formular Web. Un servlet va prelucra datele din respectivul formular şi va înregistra datele într-un document XML. Pentru vizualizarea informaţiilor existente am construit un servlet care returnează un document HTML cu datele referitoare la persoanele născute într-o anumită lună. Luna implicită este ianuarie, dar utilizatorul poate selecta luna dorită prin intermediul unui formular Web.

Codul XHTML pentru formularul de adăugare a informaţiilor este următorul:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
<head>
  <title>Adaugare informatii</title>
  <!-- Specificam setul de caractere pentru diacritice -->
  <meta http-equiv="Content-Type"
    content="text/html; charset=iso-8859-2" />
</head>

<body>
<h2 align="center">Ad&#259;ugare informa&#355;ii</h2>
<form action="http://localhost:8080/examples/servlet/Adaugare"
  method="POST">
<table border="0" align="center" cellpadding="3"
  cellspacing="10">
<tr>
  <th>Numele &#351;i prenumele:</th>
  <td><input type="text" size="30" name="nume" /></td>
</tr>
<tr>
  <th>Data na&#351;terii (zi, lun&#259;, an):</th>
  <td>
   <input type="text" size="2" maxlength="2" name="zi" />&nbsp;
   <input type="text" size="2" maxlength="2" name="luna" />&nbsp;
   <input type="text" size="4" maxlength="4" name="an" />&nbsp;
  </td>
</tr>
<tr>
  <th>Adresa:</th>
  <td><textarea name="adresa" cols="30" rows="5"></textarea>
    </td>
</tr>
<tr>
  <th>Telefon:</th>
  <td><input type="text" size="30" name="telefon" /></td>
</tr>
<tr>
  <th colspan="2">
    <input type="submit" value="&Icirc;nregistrare" />
  </th>
</tr>
</table>
```

```
</form>
</body>
</html>
```

În navigatorul Web, acest formular va arăta astfel:


Adaugare informatii - Microsoft Internet ...
File Edit View Favorites Tools Help
Back Search Favorites Links
Înregistrare
Done
My Computer

Adăugare informaţii

Numele şi prenumele: Munteanu Daniel

Data naşterii (zi, lună, an): 31 07 1979

Adresa:
Str. Clopotari Nr. 9 Bl. 930
Sc. A Et.3 Ap. 13

Telefon: 0700-129065

Acest document poate fi plasat în orice director de pe maşina curentă (locală). La apăsarea butonului de *Înregistrare* se va apela servletul `Adaugare`, care are următorul conţinut:

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.Date;
import java.util.Collection;
import java.util.Vector;
import java.util.Iterator;
import java.text.SimpleDateFormat;
import java.util.GregorianCalendar;
import java.beans.XMLDecoder;
import java.beans.XMLEncoder;

public class Adaugare extends HttpServlet {
  /** Contextul in care ruleaza servletul */
  private ServletContext context = null;

  /** Metoda de initializare
```

```
*  Se initializeaza contextul servletului
             */
public void init(ServletConfig config)
  throws ServletException {
    super.init(config);
    this.context = config.getServletContext();
}

/** Procesarea informatiilor primite prin metoda POST */
protected synchronized void doPost(HttpServletRequest cerere,
    HttpServletResponse raspuns)
      throws ServletException, java.io.IOException {

  // Stabilirea tipului de document returnat
  raspuns.setContentType("text/html");

  String mesaj = "Datele au fost inregistrate.";

  // Preluarea parametrilor
  String nume = cerere.getParameter("nume");
  String zi = cerere.getParameter("zi");
  String luna = cerere.getParameter("luna");
  String an = cerere.getParameter("an");
  String adresa = cerere.getParameter("adresa");
  String telefon = cerere.getParameter("telefon");

  // Crearea unei colectii de persoane
  Collection persons = new Vector();
  Persoana pers = null;

  try {
    String path = context.getRealPath("/");
    try {
      // mai intai se citesc datele din fisier...
      FileInputStream fis = new FileInputStream(path +
        "temp/date.xml");
      XMLDecoder decoder = new XMLDecoder(
        new BufferedInputStream(fis));
      while (null != (pers = (Persoana) decoder.readObject())) {
        // ... si se adauga in colectie
        persons.add(pers);
      }
      decoder.close(); // inchiderea fisierului
    } catch (Exception ex) { }
    // Validarea datei
    int dd = Integer.parseInt(zi);
    if ( !((0 < dd) && (dd <= 31))) {
      throw new NumberFormatException("Ziua incorecta");
    }
    int mm = Integer.parseInt(luna);
    if ( !((0 < mm) && (mm <= 12))) {
      throw new NumberFormatException("Luna incorecta");
    }
    int yyyy = Integer.parseInt(an);
    if ( !(1900 < yyyy)) {
      throw new NumberFormatException("An incorect");
    }

    // Crearea unui obiect Persoana
    SimpleDateFormat dateFormater = new SimpleDateFormat(
       "dd/MM/yyyy");
    pers = new Persoana(nume,
       dateFormater.parse(zi + "/" + luna + "/" + an),
       adresa, telefon);
    // adaugarea la colectie
    persons.add(pers);

    // suprascrierea fisierului
    FileOutputStream fos = new FileOutputStream(path +
       "temp/date.xml", false);
    XMLEncoder encoder = new XMLEncoder(new BufferedOutput
    Stream(fos));

    // salvarea datelor in fisier
    for(Iterator it = persons.iterator(); it.hasNext();) {
      encoder.writeObject((Persoana) it.next());
    }
    encoder.close();

    // Tratarea exceptiilor
  } catch(NumberFormatException e) {
     mesaj = "Data este incorecta!";
  } catch(SecurityException e) {
     mesaj = "Nu este permisa scrierea pe server!";
  } catch (Exception ex) {
     mesaj = ex.getMessage();
  }

  // Returnarea raspunsului pentru navigator
```

```
      PrintWriter iesire = raspuns.getWriter();
      iesire.println("<html><head>");
      iesire.println("<title>Rezultat adaugare</title>");
      iesire.println("</head>\n<body>");
      iesire.println("<h3>" + mesaj + "</h3>");

      // Trimitem datele primite
      iesire.println("<pre>");
      iesire.println(nume);
      iesire.println(zi);
      iesire.println(luna);
      iesire.println(an);
      iesire.println(adresa);
      iesire.println(telefon);
      iesire.println("</pre>");

      iesire.println("</body></html>");
      iesire.close(); //inchiderea fluxului de iesire
    }

    /** Ofera informatii referitoare la servlet */
    public String getServletInfo() {
      return "Adaugare data nasterii. v1.0";
    }

    /** Metoda apelata la distrugerea servletului */
    public void destroy() {
      context = null;
    }
  }
```

Pentru adăugarea unei persoane, s-au citit mai întâi datele existente în documentul XML, s-a adăugat persoana în colecţia de persoane, după care s-a salvat din nou întreaga colecţie. Acest lucru ar putea constitui un dezavantaj în cazul în care ar fi un număr mare de persoane, adăugarea realizându-se lent. O rezolvare ar fi stocarea în documente separate a persoanelor născute întro anumită lună (sau chiar într-o anumită zi din an, când numărul de persoane dintr-o lună este mare). Cititorul va modifica respectivul exemplu în funcţie de numărul de persoane pe care doreşte să le gestioneze.

Metoda `doPost()` este declarată ca fiind `synchronized` pentru a nu permite actualizarea simultană a datelor de către două fire de execuţie ale servletului (două cereri de adăugare). Altfel, ar putea apărea probleme la scrierea datelor în fişier sau pierderea de date. Dacă se încearcă vizualizarea persoanelor dintr-o anumită lună în timp ce lista acestora este actualizată, este posibil să apară o afişare necorespunzătoare sau o excepţie de intrare/ieşire. Lăsăm în sarcina cititorului implementarea restricţiei de citire a datelor în timp ce acestea sunt modificate.

Pentru a gestiona mai uşor datele despre persoane, am construit clasa `Persoana`:

```
import java.util.Date;
import java.io.Serializable;

public class Persoana implements Serializable {
  private String nume, adresa, telefon;
  private Date dataNasterii;

  /** Constructor fara parametri */
  public Persoana() {
    nume = "";
    adresa = "";
    telefon = "";
    dataNasterii = null;
  }

  /** Constructor cu parametri */
  public Persoana(String nume, Date dataNasterii,
       String adresa, String telefon) {
    this.nume = nume;
    this.dataNasterii = dataNasterii;
    this.adresa = adresa;
    this.telefon = telefon;
  }

  /* Metode pentru setarea valorilor */
  public void setNume(String nume) { this.nume = nume; }
  public void setDataNasterii(Date dataNasterii) {
    this.dataNasterii = dataNasterii;
  }
  public void setAdresa(String adresa) { this.adresa = adresa;
  public void setTelefon(String telefon) {
    this.telefon = telefon;
  }

  /* Metode pentru obtinerea informatiilor private */
  public String getNume() { return nume; }
  public Date getDataNasterii() { return dataNasterii; }
  public String getAdresa() { return adresa; }
  public String getTelefon() { return telefon; }

  /** Genereaza o linie de tabel HTML */
  public String linieTabel() {
    // se creeaza un calendar
    java.util.Calendar cal = java.util.Calendar.getInstance
    // stabilim data nasterii pentru calendar
    cal.setTime(dataNasterii);
    // returnam codul HTML pentru o linie de tabel
    return "<tr><td>" + nume + "</td><td>" +
       ((null != dataNasterii) ?
      cal.get(cal.DAY_OF_MONTH) + "" + (cal.get(cal.MONTH) + 1)
       + "" + cal.get(cal.YEAR) : "")+ "</td><td>" + adresa +
       "</td><td>" + telefon + "</td> </tr>";
  }
}
```

Pentru vizualizarea persoanelor născute într-o anumită lună, am construit un servlet `DataNasterii`. Acesta preia numărul lunii din parametrul `luna`. Dacă acesta nu este specificat, va fi aleasă luna ianuarie. După tabelul cu persoanele născute în luna stabilită am adăugat un meniu din care vom alege altă lună.

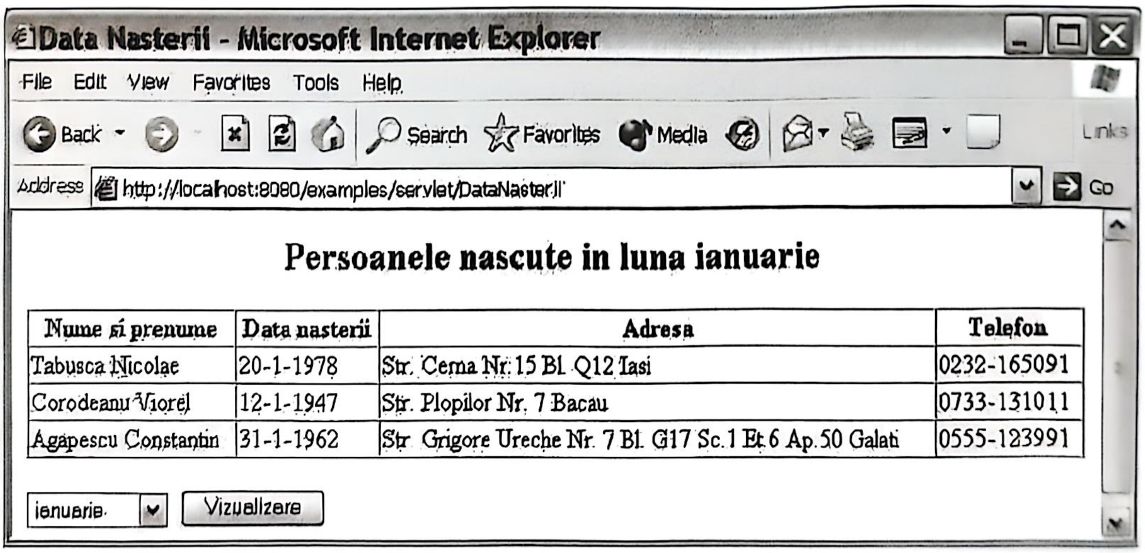
Codul corespunzător servletului `DataNasterii` este următorul:

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Calendar;
import java.beans.XMLDecoder;

public class DataNasterii extends HttpServlet {
  /** declararea datelor membre */
  protected String [] LUNA = { "ianuarie", "februarie", "martie",
     "aprilie", "mai", "iunie", "iulie", "august", "septembrie",
      "octombrie", "noiembrie", "decembrie" };
  private ServletContext context = null;

  /** metoda de initializare a servletului */
  public void init(ServletConfig config)
    throws ServletException {
    super.init(config);
    this.context = config.getServletContext();
  }

  /** metoda de procesare a cererii */
  protected void procesareCerere(HttpServletRequest cerere,
       HttpServletResponse raspuns)
       throws ServletException, java.io.IOException {
    // stabilirea tipului documentului returnat
    raspuns.setContentType("text/html");
    // Declararea variabilelor auxiliare
    String mesaj = "Ok";
    StringBuffer buf = new StringBuffer();
    Persoana pers = null;
Date data;
Calendar cal = Calendar.getInstance();
int l = 0;

try {
  // preluarea parametrului
  String luna = cerere.getParameter("luna");
  if (luna != null && luna.compareTo("")!= 0)
    l = Integer.parseInt(luna);
  // daca este luna invalida, alegem valoarea zero
  if (l < 0 && l > 11) l = 0;
  // Extragerea datelor din fisier
  try {
    String path = context.getRealPath("/");
    FileInputStream fos = new FileInputStream(path +
      "temp/date.xml");
    XMLDecoder decoder = new XMLDecoder(
      new BufferedInputStream(fos));
    // extragerea persoanelor
    while(null !=(pers =(Persoana)decoder.readObject())) 
      data = pers.getDataNasterii();
      if (null != data) {
        cal.setTime(data);
       /* Daca persoana s-a nascut in luna care prezinta
           interes, o adaugam in tabel */
        if (cal.get(cal.MONTH)== l)
           buf.append(pers.linieTabel());
      }
    }
    // inchiderea fluxului
    decoder.close();
  } catch (ArrayIndexOutOfBoundsException aie) {
  }

 // Daca nu am gasit persoane nascute in luna specificata
  if (buf.length()== 0)
    mesaj = "Nu am gasit persoane nascute in luna " +
       LUNA[l] + "!";

// tratarea exceptiilor
} catch(NumberFormatException e) {
    mesaj = "Parametru incorect!";
} catch(Exception e) {
    e.printStackTrace();
    mesaj = "Eroare interna!";
}
```

```
// scrierea raspunsului
PrintWriter iesire = raspuns.getWriter();
iesire.println("<html>");
iesire.println("<head>");
iesire.println("<title>Data Nasterii</title>");
iesire.println("</head>");
iesire.println("<body>");

// daca nu s-au inregistrat erori
if (mesaj.compareTo("Ok")==0) {
  // stabilirea titlului
  iesire.println(
      "<h2 align=\"center\">Persoanele nascute in luna "
      + LUNA[l] + "</h2>");
  // inceputul tabelului
  iesire.println("<table width=\"100%\" border=\"1\" " +
      "cellspacing= \"0\"" + "cellpading=\"3\">");
  // prima linie din tabel
  iesire.println(
      "<tr><th>Nume si prenume</th>" +
      "<th>Data nasterii </th>" +
      "<th>Adresa</th><th>Telefon</th></tr>");
  // Continutul tabelului
  iesire.println(buf);
  // sfarsitul tabelului
  iesire.println("</table>");
} else { // s-a gasit o eroare si o afisam
  iesire.println(
      "<h2 style=\"color: red; textalign: center;\">"
      + mesaj + "</h2>");
}

// formularul pentru selectarea lunii
iesire.println("<form action=\"" +
   "/examples/servlet/DataNasterii\" method=\"POST\">");
iesire.println("<select name=\"luna\">");
for (int i=0; i<12; i++)
 iesire.println("<option value=\"" + i + "\">" + LUNA[i] +
    "</option>");
iesire.println("</select>");
iesire.println(
   "&nbsp;<input type=\"submit\" value=\"Vizualizare\">");
iesire.println("</form>");

// sfarsitul documentului
iesire.println("</body>");
```

```
    iesire.println("</html>");

    // inchiderea fluxului de iesire
    iesire.close();
  }

  /** Tratarea metodei GET */
  protected void doGet(HttpServletRequest cerere,
      HttpServletResponse raspuns)
      throws ServletException, java.io.IOException {
    procesareCerere(cerere, raspuns);
  }

  /** Tratarea metodei POST */
  protected void doPost(HttpServletRequest cerere,
      HttpServletResponse raspuns)
      throws ServletException, java.io.IOException {
    procesareCerere(cerere, raspuns);
  }

  /** Informatii despre servlet */
  public String getServletInfo() {
    return "Data Nasterii. v1.0";
  }

  /** Metoda apelata la distrugerea servletului */
  public void destroy() {
    context = null;
  }
}
```

Rezultatul returnat de servletul de mai sus va fi de forma:


Data Nasterii - Microsoft Internet Explorer
File
Edit
View
Favorites
Tools
Help
Back
Search
Favorites
Media
Links
Address
http://localhost:8080/examples/servlet/DataNasterii
Go

Persoanele nascute in luna ianuarie

Nume şi prenume	Data nasterii	Adresa	Telefon
Tabusca Nicolae	20-1-1978	Str. Cerna Nr. 15 Bl. Q12 Iasi	0232-165091
Corodeanu Viorel	12-1-1947	Str. Plopilor Nr. 7 Bacau	0733-131011
Agapescu Constantin	31-1-1962	Str. Grigore Ureche Nr. 7 Bl. G17 Sc.1 Et.6 Ap.50 Galati	0555-123991

ianuarie
Vizualizare

10.4. Concluzii

Încheiem cu prezentarea unor aspecte de care trebuie să ţinem seama în lucrul cu servleturile:

1. *Performanţa.* Servleturile rulează, de obicei, în acelaşi spaţiu de proces ca şi serverul şi sunt încărcate doar o singură dată. Astfel, servleturile sunt capabile să răspundă mai rapid şi eficient la cererile clienţilor. În contrast, CGI-urile trebuie să creeze câte un nou proces pentru deservirea unei cereri. Cu toate acestea, CGI-urile scrise în limbaje compilate pot fi mai rapide.
2. *Compilarea* în *byte-code*-uri Java oferă execuţii rapide faţă de programele scrise în limbaje script. Multe dintre erori sunt îndepărtate încă de la compilare, deci servleturile sunt mai stabile. Acelaşi lucru se observă şi la CGI-urile scrise în Perl, deoarece acestea sunt compilate foarte repede, după care sunt executate.
3. *Rezistenţa la blocarea sistemului.* Maşina virtuală Java nu permite servleturilor accesul direct la locaţiile de memorie, deci se elimină posibilitatea blocării sistemului ca rezultat al accesului la memoria invalidă (pointerii în C puteau conduce la acest lucru). JVM va propaga o excepţie sau va rezolva eroarea fără blocarea sistemului.
4. *Portabilitatea platformei.* Caracteristica Java „scris o dată, rulează oriunde" permite servleturilor să fie distribuite uşor, fără a rescrie codul pentru fiecare platformă. Servleturile operează identic, fără modificări, când rulează pe UNIX, Windows sau alt sistem de operare. Acelaşi lucru este valabil şi pentru limbajele de tip script cum ar fi Perl, PHP şi bash.
5. *Portabilitatea serverelor.* Ne referim la proprietatea rulării servleturilor pe orice server Web. La adresa *http://java.sun.com/products/servlet/runners.html* există o listă completă cu serverele care furnizează suport nativ pentru servleturi în produsele lor, cum ar fi: Apache Tomcat, Oracle Application Server şi altele. Serverele JRun şi ServletExec (de la firma New Atlanta Communications) adaugă suport pentru multe servere Web, printre care: Microsoft Internet Information Server şi Apache Web Server.
6. *Durabilitate.* Servleturile rămân în memorie până când există o instrucţiune explicită de distrugere a lor. Astfel, servleturile necesită doar o instanţiere pentru a satisface mai multe cereri. De exemplu, se obişnuieşte ca la încărcarea unui servlet să se creeze şi mai multe conexiuni la baze de date.
7. *Încărcare dinamică.* Ca şi appleturile, servleturile pot fi încărcate dinamic local sau prin reţea. Încărcarea dinamică asigură neconsumarea resurselor pentru servleturile nefolosite. Ele sunt încărcate doar când este nevoie.
8. *Extensibilitatea.* Noile instrumente de dezvoltare, noile biblioteci de clase Java şi driverele de baze de date sunt constant disponibile şi astfel pot fi utilizate de servleturi. Acelaşi lucru se poate spune şi despre CGI-urile scrise în Perl.
9. *Concurenţa.* Spre deosebire de C/C++, mecanismele de concurenţă (*thread*) sunt moştenite în Java şi sintaxa sincronizării *thread*-urilor este consistentă pe

toate platformele. Natura concurenţială a servleturilor Java permite rezolvarea cererilor clienţilor în fire de execuţie separate într-un singur proces.

10. *Orientarea obiect.* Servleturile furnizează o arhitectură simplă şi elegantă pentru dezvoltarea aplicaţiilor de reţea. Asta deoarece Servlet API încapsulează toate informaţiile esenţiale şi funcţionalitatea în obiecte bine construite. De exemplu, Servlet API furnizează clase care lucrează cu obiecte abstracte, cum ar fi cereri, răspunsuri, sesiuni şi *cookie*-uri. Limbajul Perl oferă suport pentru programarea orientată obiect, însă nu este complet orientat obiect.
11. *Independenţa de protocol.* De obicei, servleturile sunt utilizate pentru extinderea funcţionalităţii serverelor Web. Servleturile sunt complet independente de protocol, acestea suportând comenzi FTP, SMTP, POP3, sesiuni Telnet, grupuri de ştiri NNTP sau orice alt protocol (fie standard, fie creat de programatori). Acelaşi lucru este valabil şi pentru alte limbaje (spre exemplu, limbajul Perl).
12. *Securitate.* Deoarece servleturile sunt scrise în Java, nu sunt posibile accesele nevalide la memorie sau violări de tip. În plus, există trei proprietăţi care fac servleturile mai sigure decât CGI-urile. În primul rând, servleturile folosesc un manager de securitate pentru crearea unor reguli de securitate. Un manager de securitate poate restricţiona reţeaua sau accesul la un fişier pentru un servlet nesigur. Pe de altă parte, un manager de securitate poate acorda drepturi depline pentru un servlet securizat. În al treilea rând, un servlet are acces la toate informaţiile conţinute în fiecare cerere a clientului. Aceste informaţii conţin date de autentificare HTTP. Când sunt utilizate în conjuncţie cu protocoalele de securitate, cum ar fi SSL, servleturile pot verifica identitatea fiecărui client. Dezavantajul este că, odată obţinute fişierele în cod binar Java (`.class`), acestea se pot decompila, ceea ce este mult mai dificil la CGI-urile scrise în C/C++.

10.5. Test-grilă

Întrebarea 10.5.1. Care dintre următoarele afirmaţii referitoare la clasa unui servlet sunt adevărate?

a) Metoda `service()` nu se poate redefini.
b) Nu se mai pot defini noi metode sau date membre.
c) Toate datele trebuie să fie publice.
d) Obligatoriu, trebuie definită una dintre metodele `doGet()` şi `doPost()`.

Întrebarea 10.5.2. Dacă un servlet redefineşte doar metoda `doGet()`, atunci:

a) Un apel prin metoda `POST` va întoarce un mesaj de eroare.
b) Metoda `doGet()` trebuie neapărat să întoarcă un rezultat.
c) În mod necesar, trebuie implementată metoda `service()`.
d) Un apel prin metoda `HEAD` va întoarce un mesaj de eroare.

Întrebarea 10.5.3. Dacă un servlet redefineşte metoda `service()`, atunci:

a) trebuie neapărat să redefinească metodele `init()` şi `destroy()`.
b) în mod necesar, va apela metodele de tipul `doXxx()`.
c) se obţine eroare la compilare.
d) nu se vor mai putea prelua parametrii transmişi de formularele Web.

Întrebarea 10.5.4. Care dintre următoarele afirmaţii sunt false?

a) Toate serverele Web posedă suport nativ pentru servleturi.
b) Pentru ca pe un calculator să se poată încărca un document Web generat de un servlet, acesta trebuie să aibă instalată maşina virtuală Java.
c) Servleturile pot fi independente de protocolul HTTP.
d) Servleturile sunt utilizate dar numai pentru prelucrarea formularelor Web.

Întrebarea 10.5.5. Rularea unui servlet se poate invoca:

a) cu ajutorul programului `java`.
b) direct dintr-un navigator Web.
c) dintr-o pagină Web, dar numai dintr-un formular.
d) cu ajutorul aplicaţiei `servletviewer`.

10.6. Exerciţii propuse spre implementare

Exerciţiul 10.6.1. Minicalculator. Scrieţi un servlet care, pentru metoda `GET`, generează un formular cu un câmp text, iar pentru metoda `POST` evaluează expresia aritmetică inserată în respectivul câmp şi întoarce un document Web cu rezultatul obţinut.

Exerciţiul 10.6.2. Agendă telefonică. Să se ofere posibilitatea consultării unei agende telefonice (căutare după nume, număr de telefon, adresă). Se va construi şi un modul pentru administrarea agendei (eliminarea, adăugarea şi, respectiv, modificarea înregistrărilor). Datele vor fi memorate într-un fişier XML.

10.7. Proiecte propuse spre implementare

Proiectul 10.7.1. Mersul trenurilor. Presupunem că dispunem de o bază de date (fie sub forma unui fişier, fie o bază de date SQL) care cuprinde mersul trenurilor din România. Scrieţi un servlet care să modeleze completarea datelor despre toate rutele dintre două oraşe (oraşul de plecare, oraşul destinaţie, numărul maxim de schimbări, ruta) şi care să furnizeze o pagină Web ce listează toate posibilităţile existente cu informaţiile aferente (timp, rută, schimbare, tip tren), sortate după numărul de ore şi minute anticipate. Se va ţine cont şi de faptul că unele trenuri nu circulă în anumite zile ale săptămânii sau în anumite perioade ale anului. Se va implementa şi un modul pentru actualizarea informaţiilor referitoare la mersul trenului (adăugarea de noi trenuri, anularea celor existente, modificarea orei de plecare, modificarea rutelor etc.).

Proiectul 10.7.2. Managementul situaţiei şcolare. Să se ofere elevilor/studenţilor oportunitatea de a consulta situaţia şcolară pe Web. Se va da posibilitatea cadrelor didactice de a nota studenţii. Bineînţeles, fiecare profesor va trebui să pună note doar la disciplina sa şi numai pentru clasele/grupele la care ţine ore. Fiecare elev/student va putea vizualiza doar notele sale. Se va da posibilitatea generării de rapoarte pentru fiecare clasă, elev/student, disciplină/profesor sau pentru întreaga instituţie (de exemplu, lista promovaţilor/nepromovaţilor).

11. Java Server Pages (JSP)

11.1. Cuvinte-cheie

- JSP, servlet, pagini Web, server-side
- JavaBeans
- Tomcat

11.2. Introducere

Java Server Pages (JSP) reprezintă una dintre cele mai puternice tehnologii Web şi este uşor de utilizat. JSP combină HTML şi XML cu servleturile şi tehnologia JavaBeans pentru a crea un mediu destul de productiv pentru dezvoltarea de situri Web independente de platformă şi de o înaltă performanţă.

Tehnologia JSP facilitează crearea conţinutului dinamic pe partea de server a paginilor Web. Este asemănătoare cu ASP (*Active Server Pages*) de pe platforma Microsoft Windows şi cu PHP (*PHP: Hypertext Preprocessor*), care este independent de platformă. JSP reprezintă o soluţie Java pentru programarea pe partea de server, fiind o alternativă la CGI-urile clasice. JSP integrează numeroase tehnologii Java cum ar fi servleturile, JavaBeans şi JDBC.

JSP extinde limbajul HTML oferind posibilitatea inserării de secvenţe de cod Java prin intermediul unor marcaje speciale. Programatorul are posibilitatea de a crea noi marcaje şi componente JavaBeans cu semnificaţiile indicate de acesta. Astfel, se pot crea noi facilităţi pentru cei care se ocupă de partea de Web design.

În acelaşi timp, JSP este o extensie a servleturilor. În loc să scriem cod Java care să genereze pagini Web, vom crea pagini Web care vor conţine elemente dinamice. Atunci când se primeşte prima cerere pentru o pagină JSP, se creează un servlet din respectiva pagină şi acesta este executat; apoi rezultatul este trimis ca răspuns la cererea primită.

Un avantaj important al JSP-urilor faţă de servleturi este faptul că se separă conţinutul HTML static de cel dinamic. În cazul servleturilor, orice modificare minoră asupra designului paginii Web implică recompilarea respectivului servlet. La JSP-uri, partea de generare a conţinutului dinamic este păstrată separat de cea statică prin utilizarea componentelor JavaBeans externe. Orice modificare a părţii statice va fi vizibilă celor ce accesează respectivul JSP, întrucât la primirea cererii se recompilează automat şi foarte repede pagina JSP, apoi se execută şi rezultatul este trimis ca răspuns la cererea primită.

Odată scrisă, o pagină JSP poate fi stocată pe orice server Web (care trebuie să aibă suport pentru JSP), oricare ar fi platforma pe care se află acesta; pagina JSP nu va suferi modificări.

Nu există limitări referitoare la tipul conţinutului generat de părţile dinamice ale JSP-urilor. Acesta poate fi text obişnuit, (X)HTML, XML, WML, SVG, X3D etc.

JSP-urile sunt mai uşor de creat şi pot avea funcţionalitatea a aproape oricărui servlet. Servleturile sunt utilizate pentru a extinde funcţionalitatea serverului Web (servicii de autentificare, validarea bazelor de date etc.) şi pentru comunicarea cu appleturi sau alte aplicaţii Web.

11.3. Elemente JSP

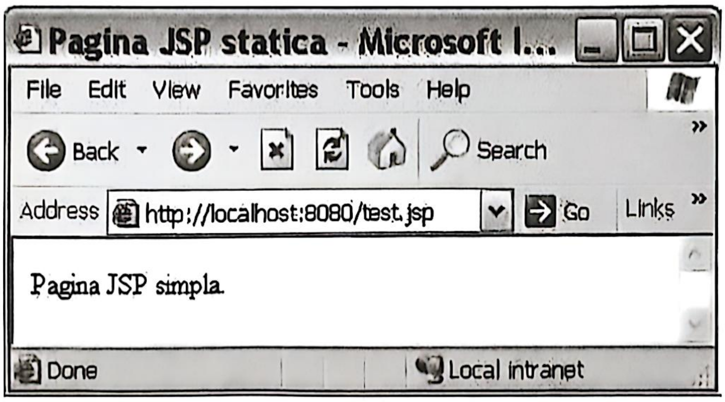
Deoarece paginile JSP sunt executate pe partea de server, avem nevoie de un server Web. Pentru testarea exemplelor vom utiliza serverul Tomcat. Paginile JSP vor fi stocate în directorul `tomcat_dir/webapps/Root`.

Paginile JSP au extensia `.jsp`. Orice pagină HTML poate fi pagină JSP.

Exemplul 11.3.1. Fie următoarea pagină simplă JSP, memorată în fişierul `test.jsp`:

```
<html>
<head>
 <title>Pagina JSP statica</title>
</head>

<body>
  <p>Pagina JSP simpla.</p>
</body>
</html>
```

Acest fişier va fi stocat în directorul mai sus amintit. Ne vom asigura că serverul Tomcat este pornit. O modalitate ar fi accesarea paginii de la adresa *http://localhost:8080/*. Apoi vizităm pagina JSP care are adresa *http://localhost:8080/test.jsp*. Rezultatul va fi:


Pagina JSP statica - Microsoft I...
File Edit View Favorites Tools Help
Back
Search
Address http://localhost:8080/test.jsp
Go
Links
Done
Local intranet

Pagina JSP simpla.

Această pagină poate fi utilizată şi ca test pentru a detecta dacă serverul suportă tehnologia JSP sau dacă este configurat corespunzător.

Exemplul 11.3.2. Următoarea pagină JSP afişează data curentă de pe serverul Web.

```
<%@ page import="java.util.Date" %>
<html>
<head>
 <title>Pagina JSP dinamica</title>
</head>

<body>
  <p>Data curenta de pe server: <b><%= new Date() %></b>.</p>
```

```
</body>
</html>
```

De asemenea, acest exemplu poate fi utilizat pentru testarea funcţionalităţii corecte a serverului Web.

11.3.1. Comentarii

Există trei tipuri de comentarii care se pot utiliza în paginile JSP: comentarii HTML, comentarii JSP şi comentarii Java.

Comentariile HTML încep cu simbolurile `<!--` şi se termină cu `-->`;. Exemplu de comentariu HTML:

```
<!-- Comentariu HTML -->
```

Aceste comentarii pot fi văzute atât de programator, cât şi de utilizatori. Utilizatorii pot vedea comentariile HTML în momentul vizualizării sursei paginii HTML. Comentariile HTML pot conţine marcaje specifice JSP care vor fi înlocuite la momentul execuţiei de către server.

Exemplul 11.3.3. Fie următoarea pagină JSP:

```
<%@ page import="java.util.Date" %>
<!-- Data curenta de pe server: <%= new Date() %>. -->
<html>
<head>
 <title>Pagina JSP dinamica</title>
</head>

<body>
  <p>Vizualizati sursa acestei pagini (View source).</p>
</body>
</html>
```

La vizualizarea sursei HTML primite de la server va apărea:

```
<!-- Data curenta de pe server: Sat Dec 09 15:23:19 EEST
2006. -->
<html>
<head>
 <title>Pagina JSP dinamica</title>
</head>

<body>
  <p>Vizualizati sursa acestei pagini (View source).</p>
</body>
</html>
```

Observăm că serverul Web nu ignoră comentariile HTML.

Comentariile JSP apar între delimitatorii `<%--` şi `--%>`. Exemplu de comentariu JSP:

```
<%-- Comentariu JSP --%>
```

Comentariile JSP sunt eliminate în momentul transformării paginii JSP în servlet şi nu vor fi vizibile utilizatorului. De aceea, se mai numesc comentarii ascunse.

Comentariile Java pot fi utilizate atunci când apar secvenţe de cod Java. Acestea nu vor apărea în paginile Web trimise de către serverul Web, deoarece sunt eliminate la transformarea paginii JSP în servlet.

Exemplul 11.3.4. Utilizarea comentariilor Java:

```
<%@ page import="java.util.Date" %>
<%!
/* data este o variabila */
  Date data = new Date();
%>
<html>
<head>
 <title>Pagina JSP dinamica</title>
</head>

<body>
  <%-- Afisam continutul variabilei 'data' --%>
  <p>Data curenta de pe server: <b><%= data %></b>.</p>
</body>
</html>
```

11.3.2. Directive

Directivele oferă posibilitatea de adăugare de informaţii adiţionale şi pentru descrierea atributelor paginii. De exemplu, directivele sunt utilizate pentru importarea pachetelor Java, includerea fişierelor şi pentru accesarea bibliotecilor de marcaje definite de utilizator.

Sintaxa generală pentru directive este:

```
<%@ directiva [...]%>
```

sau utilizând spaţiul de nume `jsp`:

```
<jsp:directive.directiva [...] />
```

Pentru claritate se poate utiliza forma XML.

11.3.2.1. *Directiva* `page`

Directiva `page` poate fi utilizată pentru a importa pachete şi clase Java, la fel ca instrucţiunea `import` (tot din Java). Exemplul 11.3.4 conţine în prima linie o directivă `page` care importă clasa `Date`, specificându-se şi pachetul din care face parte.

Directiva `page` poate avea atributele din tabelul de mai jos:

Atribut	Valoare	Valoare implicită	Descriere
`language`	`java`	`java`	specifică limbajul de programare
`extends`	superclasa	depinde de platformă	indică superclasa pentru clasa servletului care se va genera

`import`	lista de pachete separate prin virgulă	`java.lang.*,`
		`javax.servlet.http.*,`
		`javax.servlet.*,`
		`javax.servlet.jsp.*`
			importă lista de pachete şi/sau clase specificate

`session`	`true` | `false`	`true`	indică dacă se stabileşte o sesiune
`buffer`	dimensiunea în kilooctеţi	8 sau mai mult	stabileşte dimensiunea buffer-ului
`autoFlush`	`true` | `false`	`true`	indică dacă buffer-ul se goleşte automat
`isThreadSafe`	`true` | `false`	`true`	indică dacă pagina poate fi accesată de mai multe fire de execuţie simultan
`info`	text	şirul vid	specifică informaţii despre pagina JSP
`errorPage`	un URL	vidă	indică URL-ul paginii care va fi trimisă utilizatorului în caz de eroare
`isErrorPage`	`true` | `false`	`false`	indică dacă este pagină pentru raportarea erorilor
`contentType`	tipul MIME	`text/html`	specifică tipul documentului returnat

Observăm că directiva `page` stabileşte informaţiile privitoare la pagina JSP. De asemenea, remarcăm faptul că valorile implicite sunt destul de rezonabile şi rareori va trebui să le modificăm.

11.3.2.2. *Directiva* `include`

Directiva `include` conduce la includerea unui alt fişier în pagina JSP. Acest lucru este util atunci când dorim să creăm un sit unitar, în care fiecare pagină să conţină acelaşi meniu de navigare. Respectivul meniu va fi inclus într-un fişier separat (se va scrie secţiunea de cod XHTML sau JSP corespunzătoare meniului), iar acesta va fi inserat prin intermediul directivei `include` în cadrul tuturor paginilor. Dacă respectivul fişier conţine elemente JSP, acestea vor fi şi ele considerate în momentul transformării paginii JSP în servlet.

Directiva `include` are un singur atribut (`file`) care indică numele şi locul unde se găseşte respectivul fişier.

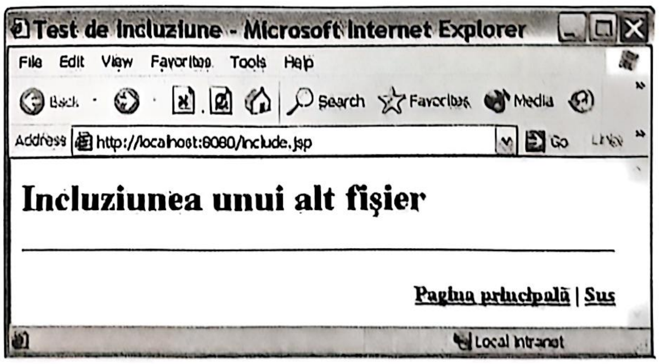
Exemplul 11.3.5. Pagina `include.jsp` va include fişierul `meniu.txt`. Ambele fişiere se află în acelaşi director. Conţinutul paginii JSP este:

```
<html>
<head>
 <title>Test de incluziune</title>
 <meta http-equiv="Content-type"
   content="text/html; charset=ISO-8859-2" />
</head>

<body>
 <a name="sus"></a>
 <h1>Incluziunea unui alt fi&#351;ier</h1>
 <%@ include file="meniu.txt" %>
</body>
</html>
```

iar al fişierului care va fi inclus:

```
<hr noshade="noshade" size="2" />
<h3 align="right">
 <a href="index.html">Pagina principal&#259;</a> |
 <a href="#sus">Sus</a>
</h3>
<%-- Comentariu JSP --%>
```

În navigatorul Web, pagina JSP va apărea astfel:



Test de incluziune - Microsoft Internet Explorer

http://localhost:8080/include.jsp

Incluziunea unui alt fişier

Pagina principală | Sus

Dacă vizualizăm pagina primită de navigator, vom observa dispariţia comentariului JSP din fişierul inclus. Acest lucru denotă faptul că fişierele incluse sunt procesate şi ele în momentul creării servletului din pagina JSP.

11.3.2.3. *Directiva* `taglib`

Crearea bibliotecilor de marcaje proprii este una dintre cele mai importante facilităţi pe care le oferă tehnologia JSP. Noi capabilităţi ale paginilor JSP se pot încapsula prin intermediul marcajelor particularizate. Acestea pot fi utilizate de către persoanele care nu au cunoştinţe de programare (cum ar fi cei care se ocupă de design). Astfel se separă partea de interfaţă de partea de funcţionalitate.

Directiva `taglib` include o bibliotecă de marcaje definite de programatori. Aceasta conţine două atribute: `uri`, care stabileşte fişierul ce conţine definiţia marcajelor, şi `prefix`, care fixează un prefix unic pentru respectiva bibliotecă de marcaje, pentru a nu apărea conflicte la utilizarea unui acelaşi marcaj care este definit în două biblioteci distincte. Prefixul se utilizează ca şi spaţiile de nume XML.

11.3.3. Declaraţii

În partea de declaraţii se pot declara date şi funcţii membre pentru utilizarea în cadrul paginilor JSP. Acestea vor face parte din servletul generat din pagina JSP. Declaraţiile se pot insera astfel:

```
<%! declaratii %>
```

sau prin

```
<jsp:declaration> declaratii </jsp:declaration>
```

În exemplul 11.3.4 am definit o variabilă de tip `Date` care conţine data curentă.

Exemplul 11.3.6. În pagina JSP de mai jos s-a definit o dată membră privată şi o metodă publică pentru obţinerea valorii datei private.

```
<%@ page import="java.util.Date" %>
<%!
  /* sectiune de declaratii */
  private static String startTime = new Date().toString();
  public static String getStartTime() { return startTime; }
%>
<html>
<head>
 <title>Pagina JSP dinamica</title>
</head>

<body>
  <p>Momentul la care JSP-ul a fost procesat de server:
    <b><%=getStartTime() %></b>.</p>
</body>
</html>
```

11.3.4. Iniţializarea şi terminarea unui JSP

Înainte de execuţie, se va apela metoda `jspInit()` pentru realizarea de iniţializări suplimentare, iar la terminarea execuţiei unui JSP se va apela metoda `jspDestroy()` pentru eliberarea unor resurse. Nici una dintre metode nu posedă parametri, ambele sunt publice şi returnează `void`.

Exemplul 11.3.7. Pagina JSP va iniţializa data membră privată `start` şi va atribui valoarea `null` la terminare.

```
<%!
 private String start="";

 public String comenteaza() {
   return "<!-- Comentariu interior -->";
 }

 public void jspInit() {
   start = "<!-- Comentariu la inceput -->";
 }

 public void jspDestroy() {
   start = null;
 }

 public String getStart() { return start; }
%>

<%= getStart() %>
<html>
<head>
 <title>Initializare JSP</title>
</head>

<body>
  <p>Continutul paginii</p>
  <%= comenteaza() %>
</body>
</html>
```

Navigatorul Web va primi următorul cod HTML:

```
<!-- Comentariu la inceput -->
<html>
<head>
 <title>Initializare JSP</title>
</head>

<body>
   <p>Continutul paginii</p>
   <!-- Comentariu interior -->
</body>
</html>
```

Se observă faptul că metoda `jspInit()` s-a apelat prima, întrucât variabila `star` a fost inițializată.

11.3.5. Obiecte implicite

Există numeroase obiecte implicite definite de arhitectura JSP. Acestea fac posibil accesul la mediul din momentul execuției. Se poate întâmpla de multe ori să nu avem nevoie în mod direct de obiectele predefinite. Lista obiectelor predefinite și o scurtă descriere a acestora se găsesc în tabelul următor:

Nume obiect	Clasa obiectului	Descriere
out	`javax.servlet.jsp.JspWriter`	Este utilizat în scriptlet-uri sau trimis ca parametru altor metode.
request	`javax.servlet.ServletRequest`	Oferă toate informațiile cu privire la cererea primită sau la navigator.
response	`javax.servlet.ServletResponse`	Oferă acces la fluxul de ieșire a servletului.
session	`javax.servlet.http.HttpSession`	Este utilizat atunci când se dorește o pseudoconexiune între client și serverul Web.
pageContext	`javax.servlet.jsp.PageContext`	Este util pentru accesarea mediului JSP și a componentelor JavaBeans.
config	`javax.servlet.ServletConfig`	Oferă informații despre proprietățile servletului.
page	`java.lang.Object`	Conține o referință la pagina JSP.
application	`javax.servlet,ServletContext`	Reprezintă aplicația Web. Acest obiect este utilizat de regulă pentru jurnalizare (engl. *logging*).
exception	`java.lang.Throwable`	Este conținută doar de paginile de eroare și conține informații privind eroarea apărută.

11.3.6. Expresii

O expresie JSP este o expresie Java, evaluată la momentul execuţiei, rezultatul fiind convertit la tipul `String` şi scris în fluxul de ieşire. Sintaxa generală a expresiilor este:

```
<%= expresie_Java %>
```

respectiv,

```
<jsp:expression> expresie_Java </jsp:expression>
```

Atenţie! Expresia nu se va termina cu simbolul punct şi virgulă.
Exemplele 11.3.2, 11.3.3, 11.3.4, 11.3.6 şi 11.3.7 conţin expresii JSP.

11.3.7. Scriptlet-uri

Până acum am văzut cum putem declara date şi funcţii membre, precum şi insera expresii. Scriptlet-urile permit adăugarea de cod Java îmbinat cu secvenţe XHTML. Sintaxa generală este următoarea:

```
<% cod_Java %>
```

respectiv,

```
<jsp:scriptlet> cod_Java </jsp:scriptlet>
```

Exemplul 11.3.8. Pagina JSP va cerceta dacă navigatorul utilizat este Internet Explorer şi va afişa un mesaj corespunzător.

```
<%
String navigator = (String) request.getHeader("user-agent");
%>
<html>
<head>
 <title>Detectare navigator</title>
</head>
<body>
<% if (navigator.indexOf("MSIE") != 1)
   {
%>
<p>Utilizati navigatorul Internet Explorer.</p>
<% } else { %>
<p> Utilizati navigatorul <%= navigator %>. </p>
<% } %>
</body>
</html>
```

Din acest exemplu se observă cum se poate intercala cod Java cu secvenţe HTML. Instrucţiunea `if` conţine cod XHTML în cele două ramuri ale sale. De remarcat că secvenţele Java şi XHTML alternează şi nu sunt imbricate.

La începutul paginii JSP se declară o variabilă locală `navigator` şi nu o dată membră.

Este recomandat să se utilizeze cât mai puţin scriptlet-urile, deoarece acestea îmbină partea de design cu cea de programare, iar pentru cei care se ocupă de design ar fi de neînţeles pagina JSP. Alternativa ar fi crearea unei biblioteci de marcaje proprii.

11.3.8. Acţiuni

Acţiunile sunt marcaje particulare predefinite. Acestea nu au corespondent un marcaj specific JSP (cele care încep cu `<%`), ci sunt numai elemente XML cu spaţiul de nume `jsp`. Acţiunile conferă un nivel înalt de funcţionalitate faţă de declaraţii, expresii şi scriptlet-uri.

Există trei categorii de acţiuni standard:

1. cele utilizate pentru componente Bean;
2. cele pentru controlul din momentul execuţiei, cum ar fi redirectarea sau includerea;
3. cele care oferă suport pentru *plug-in*-uri Java.

11.3.8.1. *Integrarea componentelor JavaBean*

Tehnologia JSP oferă o integrare foarte bună a formularelor XHTML cu componentele JavaBean. Pentru utilizarea unei componente JavaBean avem la dispoziţie acţiunile din tabelul de mai jos:

Denumirea acţiunii (marcajului)	Scopul acţiunii
`jsp:useBean`	Pregăteşte o componentă JavaBean pentru a fi utilizată în pagina JSP.
`jsp:setProperty`	Setează una sau mai multe proprietăţi pentru o componentă JavaBean.
`jsp:getProperty`	Întoarce valoarea unei proprietăţi a unei componente JavaBean sub forma unui `String`.

Acţiunea `jsp:useBean` se poate scrie ca element de sine stătător (fără conţinut):

```
<jsp:useBean id="nume" scope="scop" Specificare />
```

sau ca un element cu conţinut:

```
<jsp:useBean id="nume" scope="scop" Specificare >
  corp_pentru_creare
</jsp:useBean>
```

Această acţiune duce la crearea unui obiect corespunzător componentei JavaBean şi va avea numele `nume`.

Atributul `scope` indică durata de viaţă a componentei. Acesta poate avea valorile din tabelul următor:

`scope`	Durata existenţei componentei
`page`	Componenta este disponibilă doar în pagina JSP curentă şi va fi recreată la fiecare cerere.
`request`	Componenta este disponibilă pe tot parcursul cererii, inclusiv în paginile inserate sau redirectate.
`session`	Componenta este disponibilă pe tot parcursul sesiunii stabilite.
`application`	Componenta va fi disponibilă pentru toate sesiunile şi îşi va înceta existenţa odată cu aplicaţia Web.

Elementul *`Specificare`* este destul de flexibil şi oferă o varietate de opţiuni care sunt prezentate în tabelul de mai jos:

Opţiuni	Semnificaţie
`class="numeClasa"`	Se specifică clasa corespunzătoare componentei (`numeClasa`).
`type="numeTip"` `class="numeClasa"`	Se indică tipul care se va utiliza în pagină pentru componentă (`numeTip`), care trebuie să fie compatibil cu tipul clasei. Se indică şi numele clasei.
`type="numeTip"` `beanName="numeComp"`	Se indică tipul care se va utiliza în pagină pentru componentă (`numeTip`), precum şi numele componentei. Acesta trebuie să fie furnizat prin intermediul unei expresii JSP dată în forma JSP (cu simbolurile `<%`).
`type="numeTip"`	Se stabileşte tipul componentei, care va fi utilizat în cadrul paginii JSP.

Sintaxa acţiunii `jsp:setProperty` este următoarea:

```
<jsp:setProperty name="numeComp" extensie />
```

Numele componentei este cel stabilit de atributul `id` al acţiunii `jsp:useBean`. De exemplu, dacă avem inserată o componentă JavaBean astfel:

```
<jsp:useBean id="componenta1" ... />
```

atunci stabilirea (sau obţinerea unei proprietăţi) se realizează prin intermediul numelui `componenta1`:

```
<jsp:setProperty name="componenta1" ... />
```

Elementul *`extensie`* poate avea următoarele forme:

extensie	Descriere
`property="numeProp"` `param="numeParam"`	Stabileşte proprietatea specificată pentru parametrul indicat.
`property="numeProp"` `value="valoare"`	Setează proprietatea specificată cu valoarea dată. Valoarea trebuie să fie o expresie JSP care se evaluează în momentul execuţiei. Aceasta trebuie să fie în forma JSP (cu simbolurile `<%`).

Acţiunea `jsp:getProperty` este utilizată pentru obţinerea valorii unei proprietăţi. Aceasta are următoarea formă generală:

```
<jsp:getProperty name="numeComp" property="numeProprietate" />
```

În mod automat valoarea obţinută este convertită la tipul `String`.

11.4. Concluzii

Având la bază tehnologia servleturilor, JSP-urile permit crearea de pagini Web dinamice. Sunt mult mai uşor de implementat decât servleturile şi oferă posibilitatea de a crea aplicaţii Web complexe, putându-se utiliza componente JavaBeans, documente text sau binare, conexiuni la baze de date sau la diverse servere din reţea etc.

De cele mai multe ori, se folosesc servleturi pentru procesarea datelor de intrare, interogarea bazelor de date, efectuarea calculelor şi a operaţiilor specifice aplicaţiei Web, iar paginile JSP sunt întrebuinţate pentru afişarea rezultatelor.

Ca şi servleturile, se bucură de întregul suport oferit de limbajul Java.

11.5. Test-grilă

Întrebarea 11.5.1. Care sunt obiectele predefinite pentru o pagină JSP?

a) `out`
b) `session`
c) `servlet`
d) `exception`

Întrebarea 11.5.2. Care metode ale unui JSP sunt apelate automat?

a) `jspStart()`
b) `jspInit()`
c) `jspDestroy()`
d) `jspFialize()`

Întrebarea 11.5.3. Care dintre afirmaţiile următoare sunt adevărate?

a) Doar o singură directivă poate exista într-o pagină JSP.
b) O expresie nu poate conţine o altă expresie.
c) O expresie se termină cu simbolul „;".
d) Vizualizarea paginilor JSP se poate realiza şi fără ajutorul unui server Web.

11.6. Exerciţii propuse spre implementare

Exerciţiul 11.6.1. Să se creeze o aplicaţie Web pentru rezervarea biletelor de călătorie pentru transportul feroviar.

Exerciţiul 11.6.2. Să se construiască un sit cu anunţuri de mică publicitate. Orice utilizator poate adăuga un anunţ. Acestea sunt grupate pe categorii şi fiecare anunţ are un anumit timp de valabilitate la expirarea căruia va fi şters.

11.7. Proiecte propuse spre implementare

Proiectul 11.7.1. Să se elaboreze un sit de comerţ electronic pentru un magazin universal. Acesta va cuprinde diverse categorii de produse, oferte promoţionale, oferte de lichidări de stoc. Se va implementa un modul de căutare, coşul de cumpărături şi un modul de administrare care se va ocupa de modificarea ofertelor, actualizarea informaţiilor privitoare la produse etc. Datele vor fi memorate într-o bază de date.

Proiectul 11.7.2. Să se implementeze un sistem de *chat online*. Fiecare utilizator va avea un nume şi o parolă, pot exista mai multe camere de discuţii, se va da posibilitatea de a trimite mesaje private sau publice, de a vizualiza lista persoanelor din camera curentă, de a obţine informaţii despre un anumit utilizator etc.

12. Procesarea documentelor XML

12.1. Cuvinte-cheie

- documente XML
- XML, SAX, DOM, XSLT
- spaţii de nume XML
- validarea documentelor XML, DTD, XML Schema
- transformări, documente XSL

12.2. Standarde şi specificaţii

XML (*eXtensible Markup Language*) este un metalimbaj definit de Consorţiul Web (W3C) şi poate fi utilizat pentru a descrie o varietate de limbaje de marcare. Oferă un set de reguli, specificaţii şi convenţii pentru structurarea datelor în fişiere text (engl. *plain text*). Avantajul utilizării unui fişier text faţă de unul binar constă în faptul că programatorul sau utilizatorul pot citi, înţelege şi utiliza respectivul fişier fără a fi nevoie de programul care l-a produs. Oricum, principalii producători şi utilizatori de date XML sunt programele de calculator, şi nu utilizatorii.

Ca şi XHTML, XML utilizează elemente şi atribute. Elementele sunt cuvintele cuprinse între caracterele < şi >, iar atributele sunt secvenţe de forma `nume = "valoare"` şi apar în interiorul elementelor. XHTML-ul specifică semnificaţia fiecărui element şi atribut (adică modul de vizualizare în navigatoarele Web), în timp ce XML-ul foloseşte marcaje doar pentru a delimita datele, iar interpretarea este dată de aplicaţia care îl utilizează. Altfel spus, XML defineşte doar structura documentului, nu şi semantica acestuia.

Dezvoltarea standardului XML a fost începută în 1996 de către Consorţiul Web, în 1998 a apărut prima recomandare, revizuită în octombrie 2000, iar apoi în 2004 şi 2006. Tehnologia nu este cu totul nouă, deoarece se bazează pe SGML (*Standard Generalized Markup Language*), care a apărut în anii '80, iar în 1986 a devenit standard ISO. Creatorii XML-ului au luat părţile cele mai bune de la SGML şi au construit o tehnologie la fel de puternică ca şi SGML, dar mult mai simplă şi mai uşor de utilizat.

W3C Recommendation: Extensible Markup Language (XML) 1.0 (Fourth Edition) stabileşte sintaxa XML. Detalii despre grupul de lucru pentru XML a Consorţiului Web se găsesc la *http://www.w3.org/XML/*. Recomandarea XML 1.0 a Consorţiului Web (a patra ediţie) este disponibilă la *http://www.w3.org/TR/2006/REC-xml-20060816/*.

W3C Recommendation: Namespaces in XML 1.0 (Second edition) defineşte sintaxa şi semantica structurilor XML pentru a nu apărea conflicte între elementele cu acelaşi nume. Recomandarea pentru spaţiile de nume (engl. *namespaces*) pentru XML poate fi accesată la adresa *http://www.w3.org/TR/2006/REC-xml-names-20060816/*.

Simple API for XML Parsing (SAX) 2.0 oferă o interfaţă bazată pe evenimente pentru parsarea documentelor XML. Detalii despre SAX 2.0 se găsesc la *http://www.saxproject.org/*. În secţiunea 12.4 este prezentată modalitatea de procesare a documentelor XML utilizând SAX.

Document Object Model (DOM) Level 3 oferă un set de interfeţe definite de *W3C DOM Working Group* (Grupul de lucru DOM al W3C). Acestea descriu facilităţile pentru reprezentarea arborescentă a documentelor XML în aplicaţii. Specificaţiile acestor interfeţe sunt definite prin intermediul limbajului *Interface Definition Language* (IDL) – limbaj pentru definirea interfeţelor –, care este independent de limbajul de programare. La adresa *http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/* se găsesc specificaţiile DOM nivelul 3. Subcapitolul 12.5. descrie suportul oferit de limbajul Java pentru construirea arborelui DOM corespunzător unui document XML şi modalităţile de parcurgere a acestuia.

XSLT 1.0 (XSL Transformations) descrie un limbaj pentru transformarea documentelor XML în alte tipuri de documente, cum ar fi XML sau text. Acest limbaj este definit de Consorţiul Web, grupul de lucru pentru XSL (*eXtensible Stylesheet Language*), iar specificaţiile sunt disponibile la *http://www.w3.org/TR/1999/REC-xslt-19991116.* (De fapt, XSL este alcătuit din XSLT şi XPath.)

Pentru detalii despre tehnologiile XML consultaţi cartea *Tehnologii XML* semnată de Sabin Buraga şi apărută la Editura Polirom în 2006.

12.3. Instalare

Pachetele necesare procesării documentelor XML sunt incluse în JDK începând cu versiunea 1.4. Eventual, aceste pachete se pot procura şi instala separat, urmând indicaţiile care le însoţesc.

12.4. SAX

Simple API for XML Parsing (SAX) permite procesarea documentelor XML prin intermediul unor evenimente. Modalitatea de procesare este foarte simplă: se parcurge fişierul respectiv, iar la apariţia elementelor XML se vor lansa evenimente care pot fi tratate. Programatorul va trata evenimentele dorite şi, în funcţie de elementul (tagul) XML generator, va realiza anumite operaţii.

12.4.1. Pachete necesare

Pachetele necesare procesării documentelor XML prin intermediul SAX sunt următoarele:

- `javax.xml.parsers`
  Conţine clasele necesare procesării documentelor XML (atât cu SAX, cât şi cu DOM).
- `org.xml.sax`
  Oferă interfeţele necesare procesării documentelor XML prin SAX.
- `org.xml.sax.helpers`
  Defineşte clase ajutătoare pentru utilizarea interfeţelor SAX.
- `org.xml.sax.ext`
  Oferă două interfeţe pentru a extinde SAX-ul.

12.4.2. Exemplu de document XML

Pentru studiul procesării documentelor XML vom considera fişierul `optionale.xml`, care cuprinde informaţii referitoare la trei cursuri opţionale:

```
<?xml version="1.0" ?>
<optionale>
<optional id="C003">
  <nume>Interactiunea om-calculator</nume>
  <profesor>Sabin Corneliu Buraga</profesor>
  <an>3</an>
  <url>http://www.infoiasi.ro/~busaco/teach/interfaces/</url>
  <studenti min="15" max="60" />
  <recomandari>
    <curs>Comunicarea in medii electronice</curs>
    <curs>Grafica</curs>
  </recomandari>
</optional>

<optional id="C014">
  <nume>Compilare paralela</nume>
  <profesor>Stefan Andrei</profesor>
  <an>4</an>
  <url>http://www.infoiasi.ro/~stefan/compil/</url>
  <studenti min="15" max="80" />
  <recomandari>
    <curs>Tehnici de compilare</curs>
    <curs>Programare Java</curs>
  </recomandari>
</optional>

<optional id="C009">
  <nume>Medii virtuale</nume>
  <profesor>Stefan Ciprian Tanasa</profesor>
  <an>4</an>
  <url>http://www.infoiasi.ro/~stanasa/vr/</url>
  <studenti min="10" max="100" />
  <recomandari>
    <curs>Geometrie computationala</curs>
    <curs>Programare Java</curs>
  </recomandari>
</optional>
</optionale>
```

12.4.3. Obţinerea informaţiilor din documentele XML

În Java, fiecărui tip de eveniment îi va corespunde o metodă definită în interfaţa `ContentHandler`. Metodele sunt apelate automat de către *parser*, iar cele mai importante sunt următoarele:

- `void startDocument()`
  Este apelată automat în momentul în care se începe procesarea documentului XML.
- `void endDocument()`
  Se execută la terminarea procesării documentului.
- `void startElement(java.lang.String namespaceURI, java.lang.String localName, java.lang.String qName, Attributes atts)`
  Se apelează la apariţia unui tag. Adresa spaţiului de nume va fi transmisă în parametrul `namespaceURI` (dacă spaţiul de nume are ataşat un URI şi dacă este activată facilitatea de procesare a spaţiilor de nume). În cazul în care facilitatea de procesare a spaţiilor de nume este activă, atunci `localName` va conţine numele tagului fără prefix, iar `qName` cu tot cu prefix, altfel `localName` va conţine şirul vid, iar `qName` numele tagului. Atributele vor fi memorate în obiectul `atts`. Dacă elementul nu are atribute, atunci `atts` va fi un obiect vid.
- `void endElement(java.lang.String namespaoeURI, java.lang.String localName, java.lang.String qName)`
  Această metodă este invocată atunci când este întâlnit un tag de sfârşit, iar parametrii au aceeaşi semnificaţie ca la metoda precedentă. Dacă elementul este de sine stătător (de exemplu: `<studenti/>`), atunci metoda `endElement()` este apelată imediat după `startElement()`.
- `void characters(char[] ch, int start, int length)`
  Se execută la apariţia unui text aflat între taguri. Respectivul text va fi regăsit în tabloul `ch` începând cu poziţia `start` şi va conţine `length` caractere.
- `void processingInstruction(java.lang.String target, java.lang.String data)`
  Metoda va fi invocată la apariţia unei instrucţiuni de procesare. Parametrul `target` va conţine numele instrucţiunii de procesare, iar `data`, informaţiile aferente. De exemplu, dacă în document există instrucţiunea `<?java document = "Sax.java" ?>`, atunci în `target` se va regăsi cuvântul `java`, iar `data` va avea `document = "Sax.java"`.

Toate metodele din interfaţa `ContentHandler` pot transmite mai departe excepţia `SAXException`.

Interfaţa `ContentHandler` este implementată de clasa `DefaultHandler`. Aceasta mai implementează interfeţele `DTDHandler`, `EntityHandler` şi `ErrorHandler`. Implicit, fiecare metodă corespunzătoare interfeţelor amintite nu va avea nici un efect.

În scopul construirii unui obiect pentru efectuarea operaţiilor dorite, care se vor executa la procesarea unui document XML, va trebui să declarăm o clasă care să implementeze interfaţa `ContentHandler` sau să extindă clasa `DefaultHandler`.

Clasa `SAXParserFactory` este utilizată pentru crearea şi configurarea de procesoare bazate pe SAX. Nu este recomandat să se utilizeze mai multe instanţe ale acestei clase într-un fir de execuţie. De altfel, e necesară o atenţie deosebită în utilizarea unui astfel de obiect de către mai multe fire de execuţie. O aplicaţie poate obţine mai multe instanţe ale clasei `SAXParser` dintr-un obiect de tip `SAXParserFactory`. Pentru obţinerea unei instanţe a clasei `SAXParserFactory` se utilizează metoda statică `newInstance()` din aceeaşi clasă.

Clasa `SAXParser` este utilizată pentru procesarea efectivă a documentelor XML. Instanţele acestei clase se obţin prin apelul metodei `newSAXParser()` din clasa `SAXParserFactory`. Nu este indicată utilizarea concomitentă a unui obiect de tip `SAXParser` de către mai multe fire de execuţie. Fişierul XML poate fi dat ca un obiect de tip `File`, `InputSource`, `InputStream` sau `String` (URI). Acesta este transmis ca parametru metodei `parse()` împreună cu un obiect `DefaultHandler`.

Exemplul 12.4.1. În fişierul `Sax01.java` am definit clasa `MySAXHandler` care extinde clasa `DefaultHandler`. Clasa `MySAXHandler` este utilizată pentru procesarea fişierului `optionale.xml`. Clasa `Sax01` va conţine funcţia `main()`, care va crea obiectele necesare şi va iniţia procesarea documentului XML. Programul va avea ca efect afişarea tuturor cursurilor opţionale. Fişierul `Sax01.java` este următorul:

```
import javax.xml.parsers.*;
import org.xml.sax.helpers.*;
import org.xml.sax.*;
import java.io.*;

class MySAXHandler extends DefaultHandler {
  // indicator pentru a stabili care informatie cuprinsa
  // intre taguri va fi afisata pe ecran
  private int flag;

  // constructor fara parametri
  public MySAXHandler() { flag = 0; }

  // se apeleaza la inceputul parsarii documentului
  public void startDocument() throws SAXException {
    System.out.println("Cursurile optionale sunt:");
  }

  // se apeleaza cand se ajunge la un tag XML
  public void startElement(String ns, String local,
    String name, Attributes atts)
     throws SAXException {
    // ne intereseaza doar tagurile nume
    if (name.equals("nume")) flag = 1;
  }

  // se apeleaza cand se intalneste un tag de sfarsit
  public void endElement(String ns, String local, String name)
        throws SAXException {
```

```
      if (name.equals("nume")) flag = 0;
    }

    // se apeleaza cand se ajunge la o informatie dintre taguri
    public void characters(char[] ch, int start, int length)
      throws SAXException
    {
      if (flag == 1)
      {
        System.out.print(" * ");
        for(int i = 0; i < length; i++)
          System.out.print(ch[start+i]);
        System.out.println("");
      }
    }
}

public class Sax01 {
    public static void main(String arg[]) {
      SAXParserFactory factory = SAXParserFactory.newInstance();
      DefaultHandler handler = new MySAXHandler();
      SAXParser parser;

      try {
        parser = factory.newSAXParser();
        parser.parse("optionale.xml", handler);
      } catch (SAXException se) {
        System.err.println("SAXException:\n" + se.getMessage());
      } catch (IOException ioe) {
        System.err.println("IOException:\n" + ioe.getMessage());
      } catch (ParserConfigurationException pce) {
        System.err.println("ParserConfigurationException:\n" +
           pce.getMessage());
      }
    }
}
```

Se observă că la începutul procesării documentului se va afişa mesajul „`Cursurile optionale sunt:`". Apoi se vor considera doar tagurile `nume` care vor seta indicatorul *flag*. Când se va întâlni tagul de sfârşit, se va atribui indicatorului valoarea zero. Şirurile de caractere din afara tagurilor vor fi afişate doar dacă a fost setat indicatorul, adică se află în interiorul tagului `nume`.

Programul are ca efect afişarea următorului text:

```
Cursurile optionale sunt:
 * Interactiunea om-calculator
```

```
  * Compilare paralela
  * Medii virtuale
```

12.4.4. Obţinerea valorilor atributelor unui tag

Dacă dorim să obţinem pentru fiecare curs numărul minim, respectiv maxim de studenţi care pot urma cursul, atunci va trebui să preluăm valorile atributelor `min` şi `max` ale tagului `studenti`. Acest lucru îl obţinem prin adăugarea la exemplul 12.4.1 a următorului cod înainte de sfârşitul metodei `startElement()` din clasa `MySAXHandler`:

```
else if(name.equals("studenti"))
{
   String min = atts.getValue("min");
   String max = atts.getValue("max");
   if (min != null) // daca exista atributul min
     System.out.println("\tNumarul minim de studenti: " + min
   if (max != null) // daca exista atributul max
     System.out.println("\tNumarul maxim de studenti: " + max
}
```

Se testează mai întâi dacă s-a ajuns la tagul `studenti`, iar în caz afirmativ, se vor prelua valorile acestora prin apelul metodei `getValue()` din interfaţa `Attributes`. Acesta primeşte ca parametru numele atributului pentru care se doreşte aflarea valorii. În cazul în care atributul nu a fost specificat, atunci se întoarce valoarea `null`.

Pentru aflarea tuturor atributelor se poate proceda în felul următor: se va obţine mai întâi numărul total de atribute ale tagului curent prin utilizarea metodei `getLength()`, apoi prin intermediul unei structuri repetitive (`for`, `while`) se vor parcurge indecşii; pentru aflarea valorii unui atribut care are un anumit index, se va utiliza metoda `getValue()`, iar pentru numele atributului, `getQName()`.

De exemplu, pentru afişarea tuturor atributelor tagului `studenti` cu valorile corespunzătoare, se va înlocui codul adăugat anterior cu următorul:

```
else if(name.equals("studenti"))
{
   for(int i = 0; i < atts.getLength(); i++)
      System.out.println("\t" + atts.getQName(i) + " = " +
         atts.getValue(i));
}
```

Rularea programului rezultat va avea ca efect:

```
Cursurile optionale sunt:
  * Interactiunea om-calculator
         min = 15
         max = 60
  * Compilare paralela
         min = 15
```

```
        max = 80
  * Medii virtuale
        min = 10
        max = 100
```

12.4.5. Procesarea documentelor XML care conţin spaţii de nume

Apar situaţii în care documentele XML utilizează spaţiile de nume (*namespaces*). Acest aspect este util pentru a specifica contextul anumitor elemente XML. Într-un document XML, un spaţiu de nume este utilizat prin intermediul atributului special `xmlns`, care este eventual urmat de caracterul „:" şi apoi de numele unui prefix. Dacă nu se specifică nici un prefix, atunci este considerat ca fiind vid. Acest prefix va putea fi utilizat în definirea elementelor din interiorul tagului care conţine atributul `xmlns`. De exemplu, pentru a indica faptul că un element face parte dintr-un spaţiu de nume pentru care am stabilit prefixul `prefix`, vom utiliza construcţia `prefix:numeTag` în loc de denumirea elementului (`numeTag`).

Vom considera ca exemplu documentul `optionale_ns.xml`, acesta obţinându-se prin adăugarea unor spaţii de nume la documentul `optionale.xml` definit în secţiunea 12.4.2:

```
<?xml version="1.0" ?>
<optionale xmlns="urn:infoiasi.ro:Cursuri">
<optional id="C003">
   <nume>Interactiunea om-calculator</nume>
   <profesor>Sabin Corneliu Buraga</profesor>
   <an>3</an>
   <url>http://www.infoiasi.ro/~busaco/teach/interfaces/</url>
   <studenti min="15" max="60" />
   <recomandari>
      <curs>Comunicarea in medii electronice</curs>
      <curs>Grafica</curs>
   </recomandari>
</optional>

<optional id="C014" xmlns:Prof="urn:infoiasi.ro:Profesori">
   <nume>Compilare paralela</nume>
   <Prof:profesor>Stefan Andrei</Prof:profesor>
   <an>4</an>
   <url>http://www.infoiasi.ro/~stefan/compil/</url>
   <studenti min="15" max="80" />
   <recomandari>
      <curs>Tehnici de compilare</curs>
      <curs>Programare Java</curs>
```

```
    </recomandari>
  </optional>

  <optional id="C009" xmlns="urn:infoiasi.ro:Optional">
    <nume>Medii virtuale</nume>
    <profesor>Stefan Ciprian Tanasa</profesor>
    <an>4</an>
    <url>http://www.infoiasi.ro/~stanasa/vr/</url>
    <studenti min="10" max="100" />
    <recomandari>
      <curs>Geometrie computationala</curs>
      <curs>Programare Java</curs>
    </recomandari>
  </optional>
  </optionale>
```

Pentru a indica faptul că documentul care urmează să fie procesat conţine declaraţii ale spaţiilor de nume, se va invoca pentru un obiect de tip `SAXParserFactory` metoda `setNamespaceAware()` cu valoarea `true` pentru singurul său parametru.

Exemplul 12.4.2. Programul următor ilustrează modalitatea de procesare a informaţiilor referitoare la spaţiile de nume:

```
import javax.xml.parsers.*;
import org.xml.sax.helpers.*;
import org.xml.sax.*;
import java.io.*;

class MySAXHandler extends DefaultHandler {

  // se apeleaza cand se ajunge la un tag XML
  public void startElement(String ns, String local, String
     name, Attributes atts)
    throws SAXException
  {
    // ne intereseaza doar tagul profesor
    if (local.equals("profesor"))
       System.out.println("NS: "+ns+"\nName: "+name);
  }

  // se apeleaza cand se intalneste o specificare a unui spatiu
  // de nume
  public void startPrefixMapping(String prefix, String uri)
     throws SAXException
  {
    System.out.println(">> Prefix: "+prefix+"\tURI: "+uri);
  }

  // se invoca atunci cand se iese din domeniul
  // de valabilitate a respectivului spatiu de nume
  public void endPrefixMapping(String prefix)
    throws SAXException
  {
    System.out.println(">> End Prefix: "+prefix);
  }
}

public class Sax03bis {
  public static void main(String arg[])
  {
   SAXParserFactory factory = SAXParserFactory.newInstance();
    DefaultHandler handler = new MySAXHandler();
    SAXParser parser;

    // se invoca procesarea spatiilor de nume
    factory.setNamespaceAware(true);

    try {
      parser = factory.newSAXParser();
      parser.parse("optionale_ns.xml", handler);
    } catch (SAXException se) {
      System.out.println("SAXException:\n" +
        se.getMessage());
    } catch (IOException ioe) {
      System.err.println("IOException:\n" + ioe.getMessage());
    } catch (ParserConfigurationException pce) {
      System.err.println("ParserConfigurationException:\n" +
      pce.getMessage());
    }
  }
}
```

Execuţia programului anterior va avea ca efect afişarea la consolă a următorului text:

```
>> Prefix:       URI: urn:infoiasi.ro:Cursuri
NS: urn:infoiasi.ro:Cursuri
Name: profesor
>> Prefix: Prof URI: urn:infoiasi.ro:Profesori
NS:
Name: Prof:profesor
>> End Prefix: Prof
>> Prefix:       URI: urn:infoiasi.ro:Optional
NS: urn:infoiasi.ro:Optional
Name: profesor
>> End Prefix:
>> End Prefix:
```

Metodele `startPrefixMapping()`, respectiv `endPrefixMapping()` sunt apelate automat atunci când se întâlneşte definirea unui spaţiu de nume, respectiv atunci când se părăseşte domeniul de vizibilitate a acestuia. Acestea fac parte din interfaţa `ContentHandler`.

Metoda `startPrefixMapping()` are doi parametri de tip `String`: primul indică prefixul ataşat, iar ultimul va indica URI-ul la care se află spaţiul de nume. Observăm că, atunci când s-a definit un spaţiu de nume pentru care nu s-a stabilit un prefix acesta este considerat ca fiind vid.

Metoda `endPrefixMapping()` are un singur parametru care indică prefixul stabilit pentru respectivul spaţiu de nume.

Din rezultatul afişat observăm că, pentru un tag, mai întâi se va apela metoda pentru indicarea definirii unui spaţiu de nume şi apoi indicarea apariţiei tagului respectiv.

Referitor la parametrii metodei `startElement()`, se observă că parametrul `local` va avea tot timpul numele tagului efectiv, parametrul name va conţine numele tagului, iar în caz că acesta va fi prefixat, va conţine şi prefixul. Parametrul `ns` va conţine URI-ul spaţiului de nume din care face parte respectivul tag. Dacă nu s-a indicat faptul că se utilizează spaţii de nume, atunci parametrii `ns` şi `local` vor fi şiruri de caractere vide.

12.4.6. Procesarea documentelor XML care trebuie validate prin DTD

Pentru ca un document XML să respecte o anumită structură, se defineşte un DTD (*Document Type Definition*). Acesta specifică care este structura documentului XML (care este numele elementului rădăcină, subtagurile acestuia cu eventualele condiţii de repetare sau de obligativitate a apariţiei, lista atributelor pentru fiecare marcaj etc.).

Există două modalităţi de a se specifica DTD-ul pentru un document XML: într-un fişier extern documentului XML sau în interiorul acestuia. În cazul în care avem mai multe documente XML care au aceeaşi structură, este indicat să se definească DTD-ul într-un fişier extern, iar fiecare fişier să-l refere. Pentru a indica fişierul cu definiţia structurii documentului `optionale.xml`, vom insera în acesta, imediat după prima linie (cea care defineşte tipul documentului), următorul rând:

```
<!DOCTYPE optionale SYSTEM "cursuri.dtd">
```

Fişierul `cursuri.dtd` specifică o structură de document, iar conţinutul acestuia poate fi:

```
<!ELEMENT optionale (optional)+>
<!ELEMENT optional (nume, profesor, an, url, studenti,
  recomandari)>
<!ELEMENT nume (#PCDATA)>
<!ELEMENT profesor (#PCDATA)>
<!ELEMENT an (#PCDATA)>
<!ELEMENT url (#PCDATA)>
<!ELEMENT studenti EMPTY>
<!ELEMENT recomandari (curs)+>
<!ELEMENT curs (#PCDATA)>

<!ATTLIST optional
   id      ID      #REQUIRED
>
<!ATTLIST studenti
   min      CDATA      #IMPLIED
   max      CDATA      #IMPLIED
>
```

Vom observa că execuţia programului anterior nu va aduce schimbări. Modificăm documentul `optionale.xml` prin adăugarea unui nou curs opţional, însă cu date incomplete:

```
<optional id="C123" />
```

Rularea programului nu va aduce nici de data aceasta vreo modificare. Acest lucru înseamnă că programul nu realizează validarea documentului XML. Pentru aceasta vom înlocui conţinutul funcţiei `main()` cu următorul cod:

```
SAXParserFactory factory = SAXParserFactory.newInstance();
DefaultHandler handler = new MySAXHandler();
SAXParser parser;
XMLReader xmlReader;

factory.setValidating(true);
try {
   parser = factory.newSAXParser();
   xmlReader = parser.getXMLReader();
   xmlReader.setContentHandler(handler);
   xmlReader.parse("optionale.xml");
} catch (SAXException se) {
   System.out.println("SAXException:\n" + se.getMessage());
} catch (IOException ioe) {
   System.err.println("IOException:\n" + ioe.getMessage());
} catch (ParserConfigurationException pce) {
   System.err.println("ParserConfigurationException:\n" +
      pce.getMessage());
}
```

Clasa `MySAXHandler` este similară cu cea din exemplul 12.4.2. Vom utiliza interfaţa `XMLReader` pentru a realiza procesarea documentului XML. Obţinem o instanţă în urma apelului metodei `getXMLReader()` din clasa `SAXParser`.

Pentru stabilirea obiectului care implementează interfaţa `ContentHandler` se va utiliza metoda `setContentHandler()`. Pentru pornirea procesării se va apela metoda `parse()`, care are ca parametru un şir de caractere ce semnifică identificatorul unic al fişierului (URI-ul).

De asemenea, pentru obiectul `factory` am apelat metoda `setValidating()` care activează validarea documentului XML.

Rularea programului obţinut va genera un mesaj de eroare ce semnalează faptul că tagul `optional` este incomplet, deoarece este obligatorie indicarea celorlalte subtaguri. Deci, acum programul cercetează dacă documentul XML respectă structura specificată. În plus, mai este afişat un mesaj de avertizare care ne anunţă că nu a fost specificat un obiect care să trateze erorile şi că este utilizat unul implicit care nu reţine decât primele 10 erori.

Trebuie să definim un obiect a cărui clasă să implementeze interfaţa `ErrorHandler`. Putem să specificăm obiectul `handler`, care este o instanţă a clasei `MySAXHandler`, ce extinde clasa `DefaultHandler`. Aceasta implementează şi interfaţa `ErrorHandler`. În funcţia `main()`, înainte de procesare vom adăuga următoarea linie de cod:

```
xmlReader.setErrorHandler(handler);
```

Această modificare va face să nu mai apară nici un mesaj de eroare, deoarece clasa `DefaultHandler` implementează funcţiile specificate în interfaţa `ErrorHandler` ca funcţii vide (fără nici o instrucţiune, deci fără nici un efect). Pentru a semnala erorile ce apar va trebui să redefinim în clasa `MySAXHandler` metodele `warning()`, `error()` şi `fatalError()`, astfel:

```
public void warning(SAXParseException spe)
      throws SAXException {
   System.out.println("Warning: " + spe.getMessage());
}

public void error(SAXParseException spe)
      throws SAXException {
   String message = "Error: " + spe.getMessage();
   throw new SAXException(message);
}

public void fatalError(SAXParseException spe)
      throws SAXException {
   String message = "Fatal Error: " + spe.getMessage();
   throw new SAXException(message);
}
```

În caz de eroare se va arunca o excepţie `SAXException` cu un mesaj corespunzător. Acum execuţia programului va afişa următorul mesaj de eroare:

```
SAXException:
Error: The content of element type "optional" is incomplete,
it must match "(nume,profesor,an,url,studenti,recomandari)".
```

Astfel, se poate realiza validarea documentelor XML pentru care s-a specificat structura prin intermediul unui DTD.

Observaţie: Dacă procesorul are activată validarea documentului XML şi acesta nu are definit un DTD, atunci se va genera un mesaj de atenţionare, iar apoi se va arunca o excepţie:

```
Warning: Valid documents must have a <!DOCTYPE declaration.
SAXException: Error: Element type "optionale" is not declared.
```

12.4.7. Procesarea documentelor XML care trebuie validate via XML Schema

În secţiunea precedentă am văzut modalitatea de realizare a verificării structurii unui document XML prin DTD. În mod similar, XML Schema defineşte structura unui document XML prin intermediul unor XML-uri (în loc de DTD-uri). Faţă de DTD-uri, XML Schema utilizează tipuri de date similare cu cele din limbajele de programare şi permite o definire mult mai restrictivă. Fiind în format XML, este mult mai uşor de procesat.

De regulă, fişierele XML Schema au extensia `.xsd` (*XML Schema Definition*). În continuare prezentăm un astfel de document care descrie structura documentului XML `optionale.xml` definit în secţiunea 12.4.2.

```
<?xml version="1.0"?>
<xsd:schema xmlns="urn:infoiasi.ro:xsdOptionale"
  targetNamespace="urn:infoiasi.ro:xsdOptionale"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified">

  <!-- Definim tagul 'optionale' -->
  <xsd:element name="optionale" type="optionaleType" />

  <!-- Definim tipul (continutul) tagului 'optionale' -->
  <xsd:complexType name="optionaleType">
    <xsd:sequence>
      <xsd:element name="optional" type="optionalType"
        minOccurs="1" maxOccurs="unbounded" />
    </xsd:sequence>
  </xsd:complexType>

  <!-- Definim tipul (continutul) tagului 'optional' -->
  <xsd:complexType name="optionalType">
    <xsd:sequence>
```

```
      <xsd:element name="nume" type="xsd:string" />
      <xsd:element name="profesor" type="xsd:string" />
      <xsd:element name="an" type="anType" />
      <xsd:element name="url" type="xsd:anyURI" />
      <xsd:element name="studenti" type="studType" />
      <xsd:element name="recomandari" />
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID" use="required" />
  </xsd:complexType>

  <!-- Definim valorile care pot sa contina tagul 'an' -->
  <xsd:simpleType name="anType">
    <xsd:restriction base="xsd:unsignedInt">
      <xsd:minInclusive value="1" />
      <xsd:maxInclusive value="4" />
    </xsd:restriction>
  </xsd:simpleType>

  <!-- Definim atributele tagului 'studenti' -->
  <xsd:complexType name="studType">
      <xsd:attribute name="min" type="xsd:unsignedInt" />
      <xsd:attribute name="max" type="xsd:unsignedInt" />
  </xsd:complexType>

</xsd:schema>
```

Considerăm că schema XML de mai sus face parte din fişierul `optionale.xsd`. Pentru a indica faptul că documentul `optionale.xsd` foloseşte pentru validare schema `optionale.xsd`, vom întrebuinţa câteva atribute asociate elementului rădăcină:

```
<optionale xmlns="urn:infoiasi.ro:xsdOptionale"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:infoiasi.ro:xsdOptionale optionale.xsd">
```

Practic, se defineşte spaţiul de nume utilizat în document, iar atributul `schemaLocation` din spaţiul de nume dat de URL-ul `http://www.w3.org/2001 XMLSchema-instance` indică spaţiul de nume întrebuinţat şi locul unde este schema XML de validare – în acest caz în acelaşi director cu documentul XML.

Pentru a se realiza validarea documentului cu XML Schema va trebui ca înaintea obţinerii unui obiect de tip `SAXParser`, să activăm utilizarea spaţiilor de nume şi a validării. Apoi pentru obiectul de tip `SAXParser` vom seta proprietatea `http://java.sun.com/xml/jaxp/properties/schemaLanguage` cu valoarea `http://www.w3.org/2001/XMLSchema`.

Dacă utilizăm clasa `MySAXHandler` construită în secţiunea anterioară, programul de mai jos va realiza validarea documentului `optionale.xml` folosind schema XML `optionale.xsd`.

```
public class SaxXSchema {
  public static void main(String arg[]) {
    SAXParserFactory factory = SAXParserFactory.newInstance();
     factory.setNamespaceAware(true);
     factory.setValidating(true);
     DefaultHandler handler = new MySAXHandler();
     SAXParser parser;

     try {
       parser = factory.newSAXParser();
       parser.setProperty(
     "http://java.sun.com/xml/jaxp/properties/schemaLanguage",
          "http://www.w3.org/2001/XMLSchema");
       parser.parse("optionale.xml", handler);
     } catch (SAXException se) {
       System.out.println("SAXException:\n" + se.getMessage());
     } catch (IOException ioe) {
      System.out.println("IOException:\n" + ioe.getMessage());
     } catch (ParserConfigurationException pce) {
       System.out.println("ParserConfigurationException:\n" +
          pce.getMessage());
     }
  }
}
```

La rularea programului de mai sus se va afişa lista cursurilor opţionale.

Pentru a vedea că într-adevăr se realizează validarea, pentru al doilea curs opţional vom stabili anul `5`. Atunci se va obţine o excepţie `SAXException` emisă de metoda `error()` din clasa `MySAXHandler`, în care se atrage atenţia că valoarea maximă permisă este `4`. În această situaţie se va afişa:

```
Cursurile optionale sunt:
 * Interactiunea om-calculator
 * Compilare paralela
SAXException:
Error: cvc-maxInclusive-valid: Value '5' is not facet-valid
   with respect to maxInclusive '4' for type 'anType'.
```

Se observă din execuţia anterioară că validarea se realizează în timpul procesării documentului şi nu înainte.

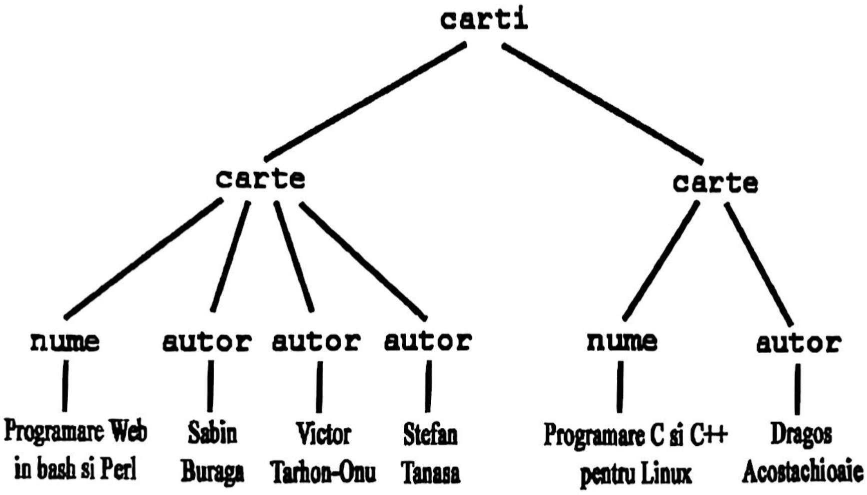
12.5. DOM

Document Object Model (DOM) este un standard stabilit de Consorţiul Web şi specifică interfeţe pentru manipularea arborelui de obiecte asociat unui document XML. Cu ajutorul DOM-ului se pot adăuga noi noduri în arbore (noi elemente sau atribute), respectiv se pot modifica şi şterge cele existente.

De exemplu, dacă avem documentul XML de mai jos:

```
<?xml version="1.0" ?>
<carti>
  <carte>
    <nume>Programare Web in bash si Perl</nume>
    <autor>Sabin Buraga</autor>
    <autor>Victor Tarhon-Onu</autor>
    <autor>Stefan Tanasa</autor>
  </carte>
  <carte>
    <nume>Programare C si C++ pentru Linux</nume>
    <autor>Dragos Acostachioaie</autor>
  </carte>
</carti>
```

atunci arborele asociat documentului XML va fi:


carti
carte
carte
nume
autor
autor
autor
nume
autor
Programare Web in bash si Perl
Sabin Buraga
Victor Tarhon-Onu
Stefan Tanasa
Programare C si C++ pentru Linux
Dragos Acostachioaie

Arborele DOM asociat unui document XML

Atributele unui element vor fi considerate ca informaţii ataşate nodului corespunzător în arborele DOM.

12.5.1. Pachete necesare

Pachetele necesare procesării documentelor XML prin intermediul DOM sunt următoarele:

- `javax.xml.parsers`
  Conţine clasele necesare procesării documentelor XML (atât cu SAX, cât şi cu DOM).
- `org.w3c.org`
  Oferă interfeţele utilizate pentru reprezentarea arborescentă a documentelor XML.

12.5.2. Crearea arborelui asociat unui document XML

Înainte de crearea unui arbore trebuie să obţinem o instanţă a unui obiect de tip `DocumentBuilderFactory`. Acesta este util pentru obţinerea de obiecte `DocumentBuilder` care, la rândul lor, permit crearea arborilor asociaţi documentelor XML.

Obţinerea unei instanţe a clasei `DocumentBuilderFactory` se realizează în urma apelului metodei statice `newInstance()` din aceeaşi clasă. Metoda `newDocumentBuilder()` întoarce o referinţă la un obiect de tip `DocumentBuilder` care posedă metoda `parse()`. Aceasta este utilizată la procesarea documentului XML şi la crearea arborelui corespunzător.

Pentru construirea arborelui corespunzător documentului `optionale.xml` vom utiliza următorul cod:

```
DocumentBuilderFactory factory =
    DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
Document document = builder.parse("optionale.xml");
```

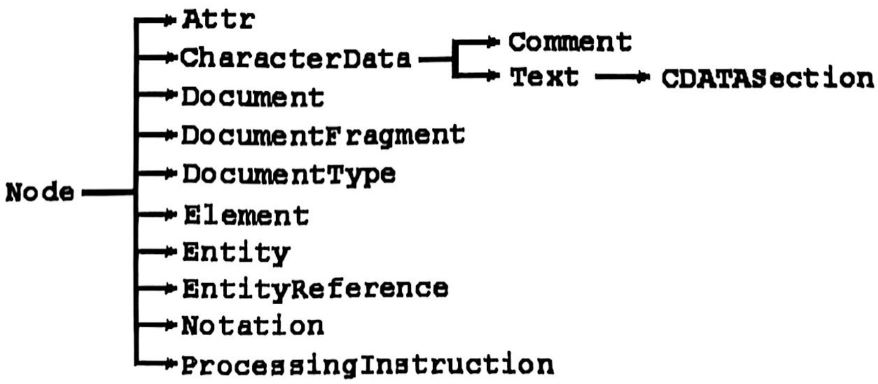
În memorie s-a construit arborele corespunzător documentului `optionale.xml`, iar referirea la acesta se va face prin intermediul interfeţei `Document`.

Nodul-rădăcină este returnat de metoda `getDocumentElement()` din interfaţa `Document`.

12.5.3. Manipularea arborelui DOM

12.5.3.1. *Interfaţa* **`Node`**

Fiecărui nod din arbore îi corespunde un obiect care implementează interfaţa `Node`. Aceasta posedă subinterfeţe corespunzătoare fiecărui tip de nod. Spre exemplu, unui element XML îi va corespunde interfaţa `Element`, pentru un comentariu `Comment` ş.a.m.d. Ierarhia de interfeţe corespunzătoare nodurilor arborelui este descrisă în figura de mai jos:



Ierarhia interfeţelor corespunzătoare nodurilor arborelui DOM

Pentru identificarea tipului nodului avem la dispoziţie metoda `getNodeType()` din interfaţa `Node`. Aceasta returnează o valoare de tip `short` care se regăseşte printre membrii statici din respectiva interfaţă:

Dată membră	Interfaţa corespunzătoare	Explicaţie
`ATTRIBUTE_NODE`	`Attr`	Nod de tip atribut
`CDATA_SECTION_NODE`	`CDATASection`	Nod corespunzător unei secţiuni CDATA
`COMMENT_NODE`	`Comment`	Nod corespunzător unui comentariu
`DOCUMENT_FRAGMENT_NODE`	`DocumentFragment`	Nod corespunzător unui fragment de document
`DOCUMENT_NODE`	`Document`	Nod corespunzător unui document XML
`DOCUMENT_TYPE_NODE`	`DocumentType`	Nod corespunzător unei secţiuni DOCTYPE
`ELEMENT_NODE`	`Element`	Nod corespunzător unui element
`ENTITY_NODE`	`Entity`	Nod corespunzător unei entităţi
`ENTITY_REFERENCE_NODE`	`EntityReference`	Nod corespunzător unei referinţe
`NOTATION_NODE`	`Notation`	Nod corespunzător unei notaţii
`PROCESSING_INSTRUCTION_NODE`	`Processing Instruction`	Nod corespunzător unei instrucţiuni de procesare
`TEXT_NODE`	`Text`	Nod de tip text

Fiecare nod are o listă cu nodurile descendente şi una cu atribute. Eventual, acestea pot fi vide.

Cele mai importante metode din clasa `Node` sunt:

- `boolean hasChildNodes()`
  Testează dacă nodul respectiv are descendenţi.
- `boolean hasAttributes()`
  Testează dacă nodul în cauză posedă atribute.
- `NodeList getChildNodes()`
  Returnează lista nodurilor descendente.
- `Node getFirstChild()`
  Returnează primul nod descendent.
- `Node getLastChild()`
  Returnează ultimul nod descendent.
- `Node getNextSibling()`
  Întoarce nodul următor celui curent din lista nodurilor descendente ale nodului-părinte.
- `Node getPreviousSibling()`
  Întoarce nodul precedent celui curent din lista nodurilor descendente ale nodului-părinte.

- `Node getParentNode()`
  Returnează nodul-părinte al nodului curent.
- `Node appendChild(Node newChild)`
  Adaugă nodul specificat la sfârşitul listei nodurilor descendente.
- `Node cloneNode(boolean deep)`
  Întoarce o copie a nodului curent (se realizează şi copierea atributelor, dacă acestea există). Dacă parametrul are valoarea `true`, atunci se realizează o copie şi pentru tot subarborele nodului respectiv.
- `Node insertBefore(Node newChild, Node refChild)`
  Inserează un nod nou (`newChild`) înaintea unui nod specificat (`refChild`) în lista nodurilor descendente. Returnează nodul inserat.
- `Node removeChild(Node oldChild)`
  Elimină din lista nodurilor descendente nodul specificat. Se returnează nodul eliminat.
- `Node replaceChild(Node newChild, Node oldChild)`
  Se înlocuieşte nodul `oldChild` cu `newChild` în lista nodurilor descendente ale nodului curent. Se returnează nodul care a fost înlocuit.
- `NamedNodeMap getAttributes()`
  Returnează o colecţie cu atributele corespunzătoare nodului curent, dacă acesta este de tip `Element`, iar în caz contrar, returnează `null`. Observăm că doar nodurile corespunzătoare elementelor au o colecţie de atribute.
- `Document getOwnerDocument()`
  Furnizează un obiect de tip `Document` corespunzător nodului curent. Orice nod împreună cu nodurile descendenţilor săi formează un arbore.
- `void normalize()`
  Uneşte nodurile de tip `Text` adiacente şi le elimină pe cele vide.
- `void setNodeValue(String newValue)`
  Modifică valoarea memorată în nodul curent, în funcţie de natura acestuia.

Valorile returnate de metodele `getNodeName()` şi `getNodeValue()` depind de natura nodurilor, după cum urmează:

Interfaţa	`getNodeName()`	`getNodeValue()`
`Attr`	numele atributului	valoarea atributului
`CDATASection`	`"#cdata-section"`	conţinutul secţiunii CDATA
`Comment`	`"#comment"`	conţinutul comentariului
`Document`	`"#document"`	`null`
`DocumentFragment`	`"#document-fragment"`	`null`
`DocumentType`	numele tipului de document	`null`
`Element`	numele elementului	`null`
`Entity`	numele entităţii	`null`

➔

`EntityReference`	numele entităţii referite	`null`
`Notation`	numele notaţiei	`null`
`ProcessingInstruction`	ţinta (engl. *target*)	conţinutul (mai puţin ţinta)
`Text`	`"#text"`	textul conţinut

Exemplul 12.5.1. Programul Java de mai jos procesează documentul `optionale.xml` (conţinutul acestuia este dat în secţiunea 12.4.2) şi afişează lista cursurilor opţionale.

```
import javax.xml.parsers.*;
import org.w3c.dom.*;

public class DOM01 {
  public static void main(String args[])
  {
    try {
      DocumentBuilderFactory factory =
         DocumentBuilderFactory.newInstance();
      DocumentBuilder builder = factory.newDocumentBuilder();

      /* Obtinerea arborelui asociat */
      Document document = builder.parse("optionale.xml");

      /* Obtinerea elementului radacina */
      Element elt = document.getDocumentElement();
      Node opt, n;
      System.out.println("Elementul radacina este: "
         + elt.getTagName());

      /* Parcurgerea arborelui */
      System.out.println("Cursurile optionale sunt:");

      /* Daca elementul-radacina nu contine elemente afisez un
         mesaj de eroare. */
      if (!elt.hasChildNodes())
      {
        System.err.println(
           "Radacina trebuie sa contina subelemente!");
        System.exit(1);
      }

      /* Obtin nodul corespunzator primului optional */
      opt = elt.getFirstChild();

      /* Cat timp mai am optionale neprocesate */
```

```
          while(opt != null)
          {
            /* Parcurgerea nodurilor descendente */
            n = opt.getFirstChild();
            while(n != null)
            /* Daca am ajuns la nodul
            corespunzator tagului nume */
            if (n.getNodeName().equals("nume"))
            {
              n = n.getFirstChild();
              System.out.print(" * ");
              while(n != null)
              {
                /* Afisez textul din interiorul tagului nume*/
                if (n.getNodeType() == Node.TEXT_NODE)
                System.out.println(n.getNodeValue().trim());
                /* trec la urmatorul nod din
                lista nodurilor descendente */
                n = n.getNextSibling();
              }
              n = null; /* iesire fortata din while */
            }
            else
              n = n.getNextSibling();
            /* trecerea la urmatorul optional */
            opt = opt.getNextSibling();
          }

        /* Prinderea exceptiilor */
        } catch (Exception e) {
           System.err.println("Exceptie: "+e.getMessage());
        }
    }
}
```

În exemplul de mai sus ne-am folosit de faptul că ştiam dinainte structura documentului XML. Acest lucru ne-a permis să adoptăm o soluţie iterativă a problemei.

Pentru obţinerea informaţiilor referitoare la nodurile arborilor sunt utilizate interfeţele `NamedNodeMap` şi `NodeList`.

Interfaţa `NamedNodeMap` este definită pentru a reprezenta o colecţie de noduri care pot fi referite cu ajutorul numelor acestora. Elementele pot fi accesate şi prin intermediul indecşilor. Este permisă eliminarea şi înlocuirea anumitor noduri. Metodele acestei interfeţe sunt prezentate în tabelul următor:

Prototipul metodei	Descrierea metodei
`int getLength()`	Întoarce numărul de elemente din colecție.
`Node item(int index)`	Returnează nodul cu indexul specificat.
`Node getNamedItem(String nume)`	Returnează nodul cu numele specificat sau `null`, în cazul în care acesta nu există.
`Node setNamedItem(Node nod) throws DOMException`	Adaugă nodul specificat la colecție. Dacă mai există un nod cu același nume, atunci acesta va fi înlocuit. În caz de succes se returnează nodul înlocuit, altfel se returnează `null`.
`Node removeNamedItem(String nume) throws DOMException`	Elimină nodul cu nume specificat din colecție. Se returnează nodul eliminat, iar în cazul în care acesta nu există în colecție se aruncă o excepție `DOMException`.
`Node getNamedItemNS(String URI, String numeLocal)`	Întoarce nodul cu numele local specificat și care se află în spațiul de nume cu URI-ul indicat. Dacă acesta nu există se obține `null`.
`Node setNamedItemNS(Node nod) throws DOMException`	Are același efect ca `setNamedItem()`, specificându-se în plus spațiul de nume din nodul specificat.
`Node removeNamedItemNS(String URI, String numeLocal) throws DOMException`	Se elimină nodul cu numele și URI-ul spațiului de nume specificat.

Interfața `NodeList` reprezintă o colecție ordonată de noduri. Se oferă suport doar pentru obținerea nodurilor din colecție, nu și pentru modificarea acesteia. Cele două metode ale interfeței `NodeList` sunt următoarele:

Prototipul metodei	Descrierea metodei
`int getLength()`	Furnizează numărul de elemente din colecție.
`Node item(int index)`	Returnează nodul cu indexul specificat. Dacă indexul este invalid, atunci se întoarce valoarea `null`.

Deși metodele interfeței `NodeList` se regăsesc printre cele din `NamedNodeMap`, nu există nici o relație de moștenire între cele două interfețe.

Exemplul 12.5.2. Programul următor va afișa valorile tuturor atributelor din documentul XML. Maniera de parcurgere a arborelui este una recursivă. Sunt utilizate interfețele `NamedNodeMap` și `NodeList` pentru parcurgerea atributelor unui nod de tip `Element`, respectiv pentru parcurgerea nodurilor descendente.

```
import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.xml.sax.*;
import java.io.*;
```

```
class UtilClass {
  public void showAttributesRec(Node n)
  {
    NodeList aux;
    NamedNodeMap map;

    /* Daca tagul are atribute atunci le vom afisa */
    if (n.hasAttributes())
    {
      System.out.println("Atributele tagului "
          + n.getNodeName() + " sunt: ");
      /* obtinem lista atributelor */
      map = n.getAttributes();
      /* parcurgem intreaga lista de atribute */
      for(int i=0; i<map.getLength(); i++)
        System.out.println("\t"+map.item(i).getNodeName()+
           "=\""+map.item(i).getNodeValue()+"\"");
      }
      /* daca avem noduri descendente */
      if(n.hasChildNodes())
      {
        aux = n.getChildNodes();
        /* parcurgem lista nodurilor descendente */
        for(int i=0; i<aux.getLength(); i++)
          /* apel recursiv */
          showAttributesRec(aux.item(i));
      }
  }
}

public class DOM02 {
  public static void main(String args[])
  {
    try {
      DocumentBuilderFactory factory = DocumentBuilderFactory.
      newInstance();
      DocumentBuilder builder = factory.newDocumentBuilder();
      Document document = builder.parse("optionale.xml");
      Element elt = document.getDocumentElement();

      /* Parcurgerea arborelui */
      UtilClass util = new UtilClass();
      util.showAttributesRec(elt);

      /* Prinderea exceptiilor */
```

```
        } catch (Exception e) {
            System.out.println("Exceptie: "+e.getMessage());
      }
    }
}
```

Am utilizat o clasă ajutătoare (`UtilClass`), în care am definit o metodă `showAttributesRec()` pentru parcurgerea recursivă a arborelui unui document. Dacă un nod posedă atribute, atunci acesta este de tip `Element` (cu siguranţă este un element), iar metoda `getNodeName()` va returna numele tagului.

În cazul în care nodul are descendenţi (alte elemente), atunci se va apela metoda `showAttributesRec()` şi pentru aceste noduri. Se observă faptul că nu se ţine cont de numele tagului pentru care se afişează atributele, de aceea programul poate procesa orice tip de document XML.

La execuţia programului anterior se va afişa la consolă:

```
Elementul radacina este: optionale
Atributele tagului optional sunt:
     id="C003"
Atributele tagului studenti sunt:
     min="15"
     max="60"
Atributele tagului optional sunt:
     id="C014"
Atributele tagului studenti sunt:
     min="15"
     max="80"
Atributele tagului optional sunt:
     id="C009"
Atributele tagului studenti sunt:
     min="10"
     max="100"
```

12.5.3.2. *Procesarea documentelor XML care utilizează spaţii de nume*

Pentru activarea facilităţii de a lua în considerare spaţiile de nume, trebuie să invocăm metoda `setNamespaceAware()` cu valoarea `true` pentru obiectul de tip `DocumentBuilderFactory`. În acest caz se vor putea utiliza metodele următoare din interfaţa `Node`:

Prototipul metodei	Descrierea metodei
`String` **`getLocalName`**`()`	Returnează numele nodului curent neprefixat de indicatorul pentru spaţiul de nume utilizat.
`String` **`getNamespaceURI`**`()`	Întoarce URI-ul spaţiului de nume curent sau `null`, dacă acesta nu este specificat.

➔

`String **getPrefix**() throws DOMException`	Returnează prefixul spaţiului de nume pentru nodul curent sau `null`, dacă acesta nu este specificat.
`void **setPrefix**(String prefix) throws DOMException`	Modifică sau elimină prefixul nodului curent. În caz de eroare se aruncă o excepţie `DOMException`.

Exemplul 12.5.3. Pentru a vedea cum sunt utilizate spaţiile de nume în arborele DOM, vom considera următorul document XML, denumit `agenti.xml`:

```
<?xml version="1.0" ?>
<agenti>
  <agent xmlns="urn:infoiasi.ro:Agent" id="dil">
    <nume nr="9">Agent Web</nume>
    <proprietate>mobil</proprietate>
  </agent>
  <a:agent xmlns:a="urn:polirom.ro:Marketing" id="PR">
    <a:nume a:nr="1">Agent</a:nume>
    <proprietate tip="generic">dinamic</proprietate>
  </a:agent>
</agenti>
```

Pentru primul marcator `agent` s-a utilizat un spaţiu de nume fără a utiliza un prefix, iar pentru cel de-al doilea s-a folosit prefixul `a`. Mai remarcăm faptul că al doilea element `proprietate` nu face parte din nici un spaţiu de nume, pe când primul este din spaţiul de nume cu URI-ul `urn:infoiasi.ro:Agent`.

Programul următor va afişa la consolă, pentru fiecare nod de tip element sau atribut, prefixul şi URI-ul spaţiului de nume corespunzător, numele şi valoarea nodului:

```
import javax.xml.parsers.*;
import org.w3c.dom.*;

public class DOMSpatiiNume {
  /* definirea unei metode private
       pentru procesarea informatiilor din arbore */
  private static void procesare(Node nod) {
   // afisare mesaj la consola
   if (nod.getPrefix() != null || nod.getNamespaceURI() != null)
      System.out.println("NS: " + nod.getPrefix() +
           "  " + nod.getNamespaceURI());
   System.out.println(nod.getNodeName() +
           "  " + nod.getNodeValue());
   System.out.println();
   // daca are atribute
```

```
      if (nod.hasAttributes()) {
        NamedNodeMap atts = nod.getAttributes();
        for(int i=0;i<atts.getLength();i++)
          procesare(atts.item(i));
      }
      // daca are descendenti
      if (nod.hasChildNodes()) {
        NodeList desc = nod.getChildNodes();
        Node n;
        for(int i=0;i<desc.getLength();i++) {
          n = desc.item(i);
          // ne intereseaza doar descendentii de tip element
          if (n.getNodeType() == n.ELEMENT_NODE)
            procesare(n);
        }
      }
    }

    public static void main(String[] args) {
      try {
        DocumentBuilderFactory factory =
            DocumentBuilderFactory.newInstance();
        // indicam faptul ca utilizam spatii de nume
        factory.setNamespaceAware(true);
        DocumentBuilder builder = factory.newDocumentBuilder();
        // crearea arborelui DOM
        Document doc = builder.parse("spatii.xml");
        // procesarea informatiilor din arbore
        procesare(doc.getDocumentElement());
      } catch (Exception e) {
        System.err.println("Exceptie: " + e);
      }
    }
  }
```

Pentru fiecare nod de tip element sau atribut se afişează maximum două linii. Dacă este utilizat un spaţiu de nume, atunci se afişează valoarea prefixului şi a URI-ului spaţiului de nume, iar ultima linie conţine numele nodului şi valoarea acestuia. Informaţiile afişate corespunzătoare nodurilor sunt separate prin linii vide.

Execuţia programului descris anterior va avea ca rezultat la consolă:

```
agenti  null

NS: null  urn:infoiasi.ro:Agent
agent  null
```

```
NS: null  http://www.w3.org/2000/xmlns/
xmlns  urn:infoiasi.ro:Agent

id  dil

NS: null  urn:infoiasi.ro:Agent
nume  null

nr  9

NS: null  urn:infoiasi.ro:Agent
proprietate  null

NS: a  urn:polirom.ro:Marketing
a:agent  null

NS: xmlns  http://www.w3.org/2000/xmlns/
xmlns:a  urn:polirom.ro:Marketing

id  PR

NS: a  urn:polirom.ro:Marketing
a:nume  null

NS: a  urn:polirom.ro:Marketing
a:nr  1

proprietate  null

tip  generic
```

Liniile care încep cu simbolurile `NS` specifică informaţiile privitoare la spaţiile de nume. Se observă faptul că primul spaţiu de nume nu are prefix, al doilea marcator `proprietate` nu este inclus în nici un spaţiu de nume ş.a.m.d.

12.5.3.3. *Procesarea documentelor XML care conţin DTD*

Pentru a indica faptul că documentul XML care se doreşte a fi procesat trebuie să fie validat conform unui DTD specificat, se va apela metoda `setValidating()` cu valoarea parametrului `true` pentru obiectul `DocumentBuilderFactory`. Implicit, această facilitate nu este activată.

Este bine să stabilim un obiect a cărui clasă implementează interfaţa `ErrorHandler` pentru a trata eventualele erori apărute la validare. Astfel, putem avea controlul validării documentelor XML. Acest lucru se realizează prin apelul metodei `setErrorHandler()` din clasa `DocumentBuilder`, ca parametru transmiţându-se obiectul care realizează tratarea erorilor.

Exemplul 12.5.4. Considerăm documentul XML din exemplul 12.5.3, în care inserăm definiţia unui DTD corespunzător după prima linie:

```
<!DOCTYPE agenti [
  <!ELEMENT agenti (agent|a:agent)+>
  <!ELEMENT agent (nume,proprietate)>
  <!ATTLIST agent
    id          CDATA      #IMPLIED
    xmlns       CDATA      #IMPLIED
  >
  <!ELEMENT a:agent (a:nume,proprietate)>
  <!ATTLIST a:agent
    id          CDATA      #IMPLIED
    xmlns:a     CDATA      #IMPLIED
  >
  <!ELEMENT nume (#PCDATA)>
  <!ATTLIST nume
    nr          CDATA      #REQUIRED
  >
  <!ELEMENT a:nume (#PCDATA)>
  <!ATTLIST a:nume
    a:nr        CDATA      #REQUIRED
  >
  <!ELEMENT proprietate (#PCDATA)>
  <!ATTLIST proprietate
    tip         CDATA      #IMPLIED
  >
]>
```

Vom denumi fişierul astfel rezultat cu `agenti_dtd.xml`. Execuţia programului Java de mai jos va afişa mesaje de eroare în cazul în care nu este respectată structura definită de DTD-ul specificat:

```
import org.w3c.dom.*;
import org.xml.sax.ErrorHandler;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;

import javax.xml.parsers.*;

public class DOMValidare {
  public static void main(String[] args) {
    try {
      DocumentBuilderFactory factory =
          DocumentBuilderFactory.newInstance();
      // indicam faptul ca utilizam spatii de nume
```

```
        factory.setNamespaceAware(true);
        /* cerem validarea documentului XML conform DTD-ului
           specificat */
        factory.setValidating(true);
        DocumentBuilder builder = factory.newDocumentBuilder();
        // indicam obiectul pentru tratarea erorilor
        MyErrorHandler err = new MyErrorHandler();
        builder.setErrorHandler(err);
        // crearea arborelui DOM
        Document doc = builder.parse("spatii_dtd.xml");
        // daca nu au fost semnalate erori
        if (!err.isEroare())
          System.out.println("OK");
      } catch (Exception e) {
        System.err.println("Exceptie: " + e);
      }
    }
}

/** clasa definita pentru tratarea erorilor */
class MyErrorHandler implements ErrorHandler {
   // declararea unei date private membre
   boolean eroare = false;

   public void error(SAXParseException e) throws SAXException {
         System.err.println("Eroare: " + e.getMessage());
         eroare = true;
   }

   public void fatalError(SAXParseException e)
      throws SAXException {
         System.err.println("Eroare fatala: " + e.getMessage());
         System.exit(1);
   }

   public void warning(SAXParseException e) throws SAXException {
         System.err.println("Avertisment: " + e.getMessage());
         eroare = true;
   }

   /** verifica daca a survenit vreo eroare */
   public boolean isEroare() { return eroare; }
}
```

Data membră `eroare` a clasei `MyErrorHandler` va indica dacă a apărut sau nu o eroare la validarea documentului XML.

În cazul în care nu apar erori, programul va afişa mesajul *OK* la consolă, altfel va apărea un mesaj de eroare. De exemplu, dacă mai adăugăm un marcator `agent` care are conţinut vid, atunci va fi afişat mesajul:

```
Eroare: Element "agent" requires additional elements.
```

care indică faptul că elementul `agent` trebuie să conţină subelemente.

12.5.3.4. *Validarea documentelor XML prin XML Schema*

Pentru DOM, validarea documentelor XML cu XML Schema se realizează la încărcarea documentului XML în memorie.

Din punct de vedere programatic, lucrurile sunt similare cu situaţia procesării prin metoda SAX. Se va activa utilizarea spaţiilor de nume şi a validării, apoi se va seta atributul `http://java.sun.com/xml/jaxp/properties/schemaLanguage` cu valoarea `http://www.w3.org/2001/XMLSchema`.

Considerăm documentele `optionale.xml` şi `optionale.xsd` utilizate în secţiunea 12.4.7 şi clasa `MyErrorHandler` definită în secţiunea precedentă pentru raportarea erorilor de validare. Aplicaţia de mai jos va realiza validarea documentului XML cu schema XML dată.

```
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.xml.sax.ErrorHandler;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;

public class DOMXSchema {
  public static void main(String args[]) {
    try {
      DocumentBuilderFactory factory =
         DocumentBuilderFactory.newInstance();
      factory.setNamespaceAware(true);
      factory.setValidating(true);

      /* Activam validarea cu XML Schema */
      factory.setAttribute(
       "http://java.sun.com/xml/jaxp/properties/schemaLanguage",
       "http://www.w3.org/2001/XMLSchema");
      DocumentBuilder builder = factory.newDocumentBuilder();

      /* Setam obiectul pentru tratarea erorilor. */
      MyErrorHandler err = new MyErrorHandler();
```

```
          builder.setErrorHandler(err);

          /* Obtinerea arborelui asociat */
          Document document = builder.parse("optionale.xml");

          /* Verificam daca a aparut vreo eroare. */
          if (!err.isEroare())
               System.out.println("OK");

          /* Obtinerea elementului radacina */
          Element elt = document.getDocumentElement();

        /* Prinderea exceptiilor */
        } catch (Exception e) {
           System.err.println("Exceptie: "+e.getMessage());
           e.printStackTrace();
        }
    }
}
```

Chiar dacă vor apărea erori de validare, arborele DOM va fi creat. Pentru a trata acest caz va trebui să ţinem cont de valoare returnată de metoda `isEroare()` din clasa `MyErrorHandler`.

Pentru a verifica dacă are loc validarea se va modifica documentul XML astfel încât acesta să nu fie conform cu schema dată, iar la execuţia programului va apărea un mesaj care să semnaleze acest lucru.

În cazul procesării documentelor XML prin metoda DOM cunoaşterea structurii documentului XML şi a restricţiilor definite reprezintă un avantaj. Astfel programatorul nu trebuie să scrie cod special pentru teste suplimentare asupra unui document XML, acest lucru revenind procesului de validare.

12.5.3.5. *Interfaţa* `Document`

Interfaţa `Document` are o importanţă deosebită, deoarece prin intermediul acesteia se pot crea noi noduri pentru arborele DOM asociat unui document XML. Aceasta mai oferă posibilitatea obţinerii unei colecţii de noduri care au acelaşi nume.

Cele mai semnificative metode ale interfeţei `Document` sunt următoarele:

- `Element getDocumentElement()`
  Întoarce nodul-rădăcină al arborelui.
- `Element getElementById(String id)`
  Returnează elementul care are identificatorul precizat sau `null`, dacă acesta nu există. Atributul `id` trebuie să fie definit ca fiind de tip `ID`.
- `NodeList getElementsByTagName(String tag)`
  Furnizează o colecţie cu toate nodurile de tip element care au numele specificat.
- `NodeList getElementsByTagNameNS(String URI, String numeLocal)`

Creează o colecţie cu nodurile de tip element din spaţiul de nume cu URI-ul dat şi cu numele indicat.

- `Node importNode(Node nod, boolean adancime) throws DOMException`
  Importă nodul specificat din alt document în cel curent. Acesta nu va avea nici un nod-părinte. Nodul-sursă nu este afectat întrucât se realizează o copia a acestuia. Dacă parametrul `adancime` are valoarea `true`, atunci se importă întreg subarborele dat de nodul importat, altfel se importă doar nodul specificat. În caz de eroare se aruncă o excepţie `DOMException`.
- `Attr createAttribute(String nume) throws DOMException`
  Creează un nod de tip atribut cu numele specificat. Acesta se poate adăuga la un nod de tip element cu ajutorul metodei `setAttributeNode()`.
- `CDATASection createCDATASection(String data) throws DOM Exception`
  Creează un nod de tip secţiune CDATA cu conţinutul specificat.
- `Comment createComment(String data)`
  Creează un nou nod de comentariu cu textul dat de parametrul `data`.
- `DocumentFragment createDocumentFragment()`
  Creează un nod vid de tip `DocumentFragment`.
- `Text createTextNode(String data)`
  Construieşte un nod text.
- `Element createElement(String numeTag) throws DOMException`
  Întoarce un nou nod corespunzător marcatorului cu numele specificat.
- `EntityReference createEntityReference(String nume) throws DOMException`
  Întoarce un nou nod de tipul `EntityReference`.
- `ProcessingInstruction createProcessingInstruction(String target, String data) throws DOMException`
  Creează un nod corespunzător unei instrucţiuni de procesare cu date indicate.

Nodurile create de metodele interfeţei `Document` nu sunt automat adăugate în arborele DOM corespunzător documentului XML. Acestea trebuie inserate în lista nodurilor descendente ale unui nod. Modificarea arborelui nu implică actualizarea fişierului-sursă XML.

Exemplul 12.5.5. Fie următorul document XML:

```
<?xml version="1.0" ?>
<!DOCTYPE studenti [
  <!ELEMENT studenti (student)+>
  <!ELEMENT student  (nume,an,grupa)>
  <!ELEMENT nume    (#PCDATA)>
  <!ELEMENT an      (#PCDATA)>
  <!ELEMENT grupa   (#PCDATA)>
]>
<studenti>
```

```
<student>
  <nume>Popovici Paul</nume>
  <an>2</an>
  <grupa>3</grupa>
</student>
<student>
  <nume>Farcas Dumitru</nume>
  <an>2</an>
  <grupa>1</grupa>
</student>
<student>
  <nume>Cristea Marius</nume>
  <an>3</an>
  <grupa>1</grupa>
</student>
<student>
  <nume>Munteanu Daniel</nume>
  <an>2</an>
  <grupa>3</grupa>
</student>
<student>
  <nume>Cracana Roxana</nume>
  <an>3</an>
  <grupa>1</grupa>
</student>
</studenti>
```

Pentru adăugarea unui nou student vom scrie următoarea secvenţă de cod:

```
/* adaugarea unui nou student */
Node stud = doc.createElement("student");
// cream numele
Node n = doc.createElement("nume");
Node t = doc.createTextNode("Mironescu Adrian");
n.appendChild(t);
stud.appendChild(n);
// cream anul
n = doc.createElement("an");
t = doc.createTextNode("3");
n.appendChild(t);
stud.appendChild(n);
// cream grupa
n = doc.createElement("grupa");
t = doc.createTextNode("2");
n.appendChild(t);
stud.appendChild(n);
```

```
// adaugarea nodului student in arborele DOM
doc.getDocumentElement().appendChild(stud);
```

Pentru a crea marcatorul `nume` cu textul *Mironescu Adrian* trebuie să creăm două noduri: unul corespunzător elementului, iar celălalt pentru nodul de tip text. Nodul tagului va avea ca descendent nodul cu text. Analog, se creează nodurile corespunzătoare marcatorilor `an` şi `grupa`. Aceşti trei marcatori vor fi descendenţi pentru nodul `student`.

Extragerea numelor tuturor studenţilor din arborele DOM se realizează astfel:

```
/* afisarea tuturor studentilor */
System.out.println("\t Lista tuturor studentilor:");
// obtinerea listei de noduri corespunzatoare tagurilor nume
NodeList lista = doc.getElementsByTagName("nume");
// parcurgerea listei
for(int i=0;i<lista.getLength();i++)
  // afisarea continutului nodului descendent de tip text
  System.out.println(lista.item(i).getFirstChild().getNodeValue());
```

Efectul execuţiei codului anterior va fi:

```
        Lista tuturor studentilor:
Popovici Paul
Farcas Dumitru
Cristea Marius
Munteanu Daniel
Cracana Roxana
Mironescu Adrian
```

Cu ajutorul arborelui DOM asociat putem modifica dinamic conţinutul documentului XML. Se pot adăuga, modifica, respectiv şterge elemente, comentarii, texte etc.

12.6. XSLT

Transformările XSLT (engl. *XSL Transformation*) descriu un limbaj pentru transformarea documentelor XML în alte documente XML sau în alte documente de tip text. Acesta este definit de grupul de lucru pentru XSL al Consorţiului Web.

Transformarea se defineşte prin intermediul unui document XSL, care este un tip particular de document XML. De regulă, ele sunt păstrate în fişiere cu extensia `.xsl`.

În acest capitol vom considera documentul XML `optionale.xml` din secţiunea 12.4.2 ca document-sursă ce urmează să fie transformat.

12.6.1. Pachete necesare

Pachetele necesare transformării documentelor XML sunt următoarele:

- `javax.xml.transform`

Conţine clasele şi interfeţele necesare transformării documentelor XML.

- `org.xml.transform.dom`
  Oferă clasele necesare specificării sursei de intrare sau ieşirii prin DOM.
- `org.xml.transform.sax`
  Oferă clasele necesare specificării sursei de intrare sau ieşirii prin SAX.
- `org.xml.transform.stream`
  Oferă clasele necesare specificării sursei de intrare sau ieşire prin intermediul fluxurilor de intrare/ieşire obişnuite.

12.6.2. Realizarea unor transformări

Pentru a realiza transformări vom recurge la clasa `Transformer`. Înainte de crearea obiectelor din această clasă, va trebui să obţinem un generator de transformări, adică o instanţă la clasa `TransformerFactory`. Acest lucru se realizează prin apelul metodei statice `newInstance()` din aceeaşi clasă. Ambele clase sunt abstracte şi fac parte din pachetul `javax.xml.transform`.

Obţinerea unei instanţe la clasa `Transformer` se realizează prin apelul metodei `newTransformer()` din clasa `TransformerFactory`. Aceasta are două forme: fără parametru, caz în care se va realiza o copie a sursei de intrare, sau cu un parametru de tip `Source`, care va referi documentul cu definiţia transformării.

Transformarea efectivă a unui document XML în alt tip de document este realizată de către metoda `transform()` din clasa `Transformer`. Prototipul metodei este următorul:

- `public abstract void transform(Source sursaXML, Result rezultat) throws TransformerException`

unde `sursaXML` desemnează documentul sau porţiunea de document XML care va fi transformată, iar `rezultat` indică unde este afişat rezultatul transformării.

`Source` şi `Result` sunt două interfeţe din pachetul `javax.xml.transform` utilizate pentru specificarea sursei de intrare, respectiv ieşirea transformării. Ambele au aceleaşi două metode:

Prototipul metodei	Descrierea metodei
`public String` **`getSystemId`**`()`	Returnează identificatorul stabilit cu metoda `setSystemId()` sau `null`, dacă aceasta nu s-a apelat.
`public void` **`setSystemId`**`(String id)`	Stabileşte identificatorul pentru sursa de intrare, respectiv ieşirea. Este utilizată pentru rezolvarea URI-urilor relative şi în mesajele de eroare sau de avertizare.

Există trei clase care implementează interfaţa `Source`. Acestea sunt `DOMSource`, `SAXSource` şi `StreamSource`. De asemenea, există trei clase care implementează interfaţa `Result`: `DOMResult`, `SAXResult` şi `StreamResult`.

Sursele de intrare sunt date de interfaţa `Source`. Există trei tipuri de surse de intrare: obişnuite, surse DOM şi surse SAX, iar pentru fiecare există câte o clasă predefinită care implementează interfaţa amintită.

Sursele obişnuite sunt specificate prin intermediul clasei `StreamSource`. Aceasta se află în pachetul `javax.xml.transform.stream`, iar cei mai semnificativi constructori ai săi sunt următorii:

Prototipul constructorului	Descrierea constructorului
`public StreamSource()`	Constructorul implicit creează un document vid.
`public StreamSource(File f)`	Se construieşte o sursă de intrare dintr-un `File` (fişier local).
`public StreamSource (InputStream flux)`	Se construieşte o sursă de intrare dintr-un `InputStream`.
`public StreamSource(Reader r)`	Se construieşte o sursă de intrare dintr-un flux `Reader`.
`public StreamSource(String url)`	Se construieşte o sursă de intrare din URL-ul specificat.

Această clasă pune la dispoziţie stabilirea sursei de intrare dintr-un `InputStream` sau `Reader`, respectiv obţinerea acestora prin intermediul metodelor următoare:

- `public void setInputStream(InputStream flux)`
- `public void setReader(Reader r)`
- `public InputStream getInputStream()`
- `public Reader getReader()`

Exemplul 12.6.1. Programul Java va afişa pe ecran lista cursurilor opţionale şi a titularilor acestora. Informaţiile sunt extrase din documentul `optionale.xml` definit în secţiunea 12.4.2, iar definiţia transformării se află în fişierul `optionale_1.xsl`. Conţinutul fişierului `optionale_1.xsl` este următorul:

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="text" />
<xsl:template match="/">
<xsl:for-each select=".//optional">
  <xsl:value-of select="nume" />,
  <xsl:value-of select="profesor" />.
</xsl:foreach>
</xsl:template>
</xsl:stylesheet>
```

iar a fişierului `VizualizareCursuri.java` este:

```
import javax.xml.transform.*;
import javax.xml.transform.stream.*;
public class VizualizareCursuri {
  public static void main(String[] args) {
      TransformerFactory factory = TransformerFactory.
        newInstance();
      Transformer trans;
      try {
      // crearea transformarii
      trans = factory.newTransformer(new StreamSource
        ("optionale_1. xsl"));
      // crearea sursei din fisierul optionale.xml
      StreamSource sursa = new StreamSource("optionale.xml");
      // rezultatul va fi afisat pe ecran
      StreamResult rezultat = new StreamResult(System.out);
      // se realizeaza transformarea
      trans.transform(sursa, rezultat);
      } catch (TransformerConfigurationException e) {
        System.err.println("Eroare la initializare!\n" + e);
      } catch (TransformerException e) {
        System.err.println("Eroare la transformare!\n" + e);
      }
    }
}
```

Transformarea definită în fişierul `optionale_1.xsl` specifică faptul că sunt selectate toate elementele `optional`, iar pentru fiecare vor fi afişate informaţiile din interiorul subtagurilor `nume` şi `profesor`.

Programul anterior va afişa în urma execuţiei la consolă:

```
Interactiunea om-calculator,
  Sabin Corneliu Buraga.
Compilare paralela,
  Stefan Andrei.
Medii virtuale,
  Stefan Ciprian Tanasa.
```

Sursele DOM sunt reprezentate de clasa `DOMSource`. Aceasta permite ca datele de intrare să fie preluate dintr-un nod al documentului XML. Transformarea se va aplica întregului subarbore determinat de nodul specificat.

Clasa `DOMSource` face parte din pachetul `javax.xml.transform.dom` şi are doi constructori importanţi:

- `public DOMSource()`
- `public DOMSource(Node nod)`

Primul este fără parametri, iar al doilea are un parametru de tip `Node`, care indică nodul din arborele DOM asociat documentului XML care va fi transformat. Se oferă posibilitatea stabilirii ulterioare a nodului prin intermediul metodei `setNode()`, precum şi obţinerea acestuia cu metoda `getNode()`.

Exemplul 12.6.2. Se va afişa pe ecran un document XML care va cuprinde doar primul opţional. Transformarea este definită în fişierul `copiere.xsl`, care are următorul conţinut:

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output indent="yes" />
<xsl:template match="@*|node()">
  <xsl:copy>
    <xsl:apply-templates select ="@*|node()" />
  </xsl:copy>
</xsl:template>
</xsl:stylesheet>
```

Programul Java este următorul:

```
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.*;
import org.w3c.dom.*;

public class AfisareNod {
  public static void main(String[] args) {
    TransformerFactory tFactory =
       TransformerFactory.newInstance();
    DocumentBuilderFactory docFactory =
       DocumentBuilderFactory.newInstance();
    try {
      // obtinerea arborelui DOM
      DocumentBuilder builder =
        docFactory.newDocumentBuilder();
      Document document = builder.parse("optionale.xml");
      // obtinerea nodului corespunzator primului optional
      Node nod = document.getDocumentElement().
         getElementsByTagName("optional").item(0);
      // crearea transformarii
      Transformer trans = tFactory.newTransformer(
         new StreamSource("copiere.xsl"));
      // crearea sursei dintr-un nod DOM
      DOMSource sursa = new DOMSource(nod);
      // rezultatul va fi afisat pe ecran
      StreamResult rezultat = new StreamResult(System.out);
      // se realizeaza transformarea
      trans.transform(sursa, rezultat);
```

```
        } catch (Exception e) {
          System.err.println("Eroare!\n" + e);
        }
    }
}
```

Transformarea dată poate fi utilizată ori de câte ori se dorește scrierea într-un flux de ieșire a unui document XML sau a unei porțiuni a lui.

Execuția programului va afișa la consolă următorul document XML:

```
<?xml version="1.0" encoding="UTF8"?>
<optional id="C003">
  <nume>Interactiunea om-calculator</nume>
  <profesor>Sabin Corneliu Buraga</profesor>
  <an>3</an>
  <url>http://www.infoiasi.ro/~busaco/courses/piu/</url>
  <studenti min="15" max="60"/>
  <recomandari>
    <curs>Comunicarea in medii electronice</curs>
    <curs>Grafica</curs>
  </recomandari>
</optional>
```

12.7. Concluzii

Limbajul Java oferă un foarte bun suport pentru procesarea documentelor XML. Sunt disponibile ambele modalități de procesare: SAX și DOM. Fiecare prezintă avantaje și dezavantaje, utilizarea acestora fiind în funcție de natura documentelor XML și de cerințele pe care trebuie să le îndeplinească aplicația.

SAX este utilizat, în general, pentru documente de mari dimensiuni și care nu necesită accesul aleatoriu la date. Parcurgerea este secvențială, de la început până la sfârșit, și nu presupune memorarea întregului fișier la un moment dat.

DOM este frecvent întâlnit în aplicațiile care necesită accesarea informațiilor din diferite porțiuni ale documentului XML. Memoria ocupată de arborele asociat nu va fi un dezavantaj foarte mare în acest caz. Din cauza consumului mare de memorie, se are în vedere ca documentele XML să nu fie de mari dimensiuni.

XSLT permite transformarea datelor XML în alte date XML sau text. De regulă, transformările sunt definite în documente separate, sunt tot XML și au extensia `.xsl`.

Pentru a defini un tip de document se pot utiliza documente DTD sau XML Schema. Acestea indică structura unui document XML și eventuale constrângeri. Pentru a realiza validarea datelor XML, la procesare va trebui activată această facilitate. De asemenea, trebuie specificat dacă se dorește procesarea unui document care conține spații de nume. Pentru validare cu XML Schema se vor seta suplimentar un atribut (pentru DOM) sau o proprietare (pentru SAX).

12.8. Test-grilă

Întrebarea 12.8.1. Care dintre afirmaţiile următoare sunt adevărate?
a) Documentele XML sunt binare.
b) Procesarea documentelor XML se poate realiza doar utilizând limbajul Java.
c) Structura unui document XML poate fi specificată într-un fişier extern.
d) Utilizând SAX nu se pot procesa documente XML de mari dimensiuni.

Întrebarea 12.8.2. Clasa `DefaultHandler` implementează interfeţele:
a) `ErrorHandler`
b) `TagHandler`
c) `ContentHandler`
d) `SAXHandler`

Întrebarea 12.8.3. Care sunt superinterfeţele interfeţei `Text`?
a) `CDATASection`
b) `CharacterData`
c) `Comment`
d) nu există această interfaţă

Întrebarea 12.8.4. Care sunt subinterfeţele interfeţei `Node`?
a) `NodeList`
b) `ProcessingInstruction`
c) `Document`
d) `NamedNodeMap`

Întrebarea 12.8.5. Care dintre afirmaţiile referitoare la arborele DOM sunt false?
a) Nu se pot adăuga noduri aparţinând altui arbore DOM.
b) Se pot elimina subarbori.
c) Se pot înlocui nodurile existente cu altele noi.
d) Nu pot exista două noduri adiacente de acelaşi tip şi cu acelaşi nume.

12.9. Exerciţii propuse spre implementare

Exerciţiul 12.9.1. Să se implementeze o clasă care oferă posibilitatea eliminării anumitor elemente şi atribute dintr-un document XML. Pentru a testa funcţionalitatea respectivei clase se va crea o aplicaţie în care utilizatorul va specifica prin intermediul unui formular elementele şi atributele care vor fi eliminate.

Exerciţiul 12.9.2. Să se elaboreze un document XML cu informaţii privitoare la orarul săptămânal (se va construi şi un DTD pentru validare). Se va realiza o aplicaţie pentru vizualizarea orarului şi o unealtă pentru actualizarea acestuia.

Exercițiul 12.9.3. Să se construiască un arbore DOM corespunzător arborelui genealogic. Vizualizarea grafică va fi realizată de către un applet.

12.10. Proiecte propuse spre implementare

Proiectul 12.10.1. Să se realizeze un program care pentru orice document XML generează un DTD care să fie cel mai restrictiv. Eventual se va extinde aplicația pentru a genera un DTD pentru mai multe documente XML similare.

Proiectul 12.10.2. Să se construiască o aplicație care gestionează biblioteca personală. Datele vor fi stocate în fișiere XML și se vor implementa următoarele facilități:

1. adăugarea/ștergerea unei cărți din evidență;
2. modificarea informațiilor referitoare la o anumită carte;
3. căutarea după unul sau mai multe criterii (autor, titlul cărții, editură, anul apariției);
4. elaborarea unui raport referitor la cărțile împrumutate (se vor oferi și informații despre persoana căreia i s-a împrumutat cartea).

13. Interacţiunea cu utilizatorul prin Swing

13.1. Cuvinte-cheie

- interfeţe grafice
- componente
- evenimente
- ascultători
- modele
- *look-and-feel*

13.2. Introducere

Biblioteca Swing este partea JFC (engl. *Java Foundation Classes*) care oferă componentele necesare programatorilor pentru crearea de interfeţe grafice moderne, complexe şi într-adevăr prietenoase aplicaţiilor Java. Swing aduce îmbunătăţiri calitative faţă de mai vechiul pachet AWT (engl. *Abstract Window Toolkit*), folosit în trecut pentru realizarea interfeţelor grafice, completându-i neajunsurile. Spre exemplu, în distribuţia AWT pentru Java 1.0, programatorul avea la dispoziţie numai patru fonturi cu care putea să opereze şi un număr restrâns de tipuri de componente, insuficiente pentru crearea de interfeţe cu un grad ridicat de complexitate. Totuşi pachetul Swing nu este total separat de AWT cum s-ar putea crede, ci se poate spune că este construit peste acesta (engl. *on top*), aşa cum se va observa din cuprinsul acestui capitol, AWT fiind folosit mai mult pe post de intermediar între Swing şi sistemul de operare peste care este instalată platforma Java.

Printre îmbunătăţirile aduse de Swing, în afară de noile componente adăugate, putem enumera modul de prezentare şi comportare pentru interfaţă independentă de platformă, posibilitatea de a avea componente cu forme nerectangulare, accesibilitate pentru persoanele cu deficienţe etc. Acest pachet a fost astfel gândit, încât pentru lucruri simple este nevoie de scrierea de cod puţin, iar pentru lucruri mai complexe, mărimea codului creşte proporţional. De asemenea, se folosesc convenţii logice pentru denumirea claselor, metodelor etc. care, odată înţelese şi învăţate, determină creşterea vitezei de lucru.

13.2.1. Obiectivele urmărite în acest capitol

Datorită faptului că dezbaterea unui subiect precum Swing necesită spaţiul unei cărţi întregi, şi nu al unui singur capitol, am încercat să surprindem în primul rând principiile fundamentale care stau la baza acestui pachet într-adevăr complex, a căror înţelegere dă posibilitatea folosirii Swing la adevărata lui valoare. De asemenea, vom descrie fiecare componentă în parte, prezentându-i principalele caracteristici, cum ar fi localizarea componentei în cadrul pachetului Swing, modalităţi de construcţie a componentelor, modalităţi de personalizare, evenimentele pe care componenta le poate genera, respectiv un exemplu de folosire efectivă a ei într-o aplicaţie având codul comentat, îndeajuns pentru ca lectorii să pornească propriile lor implementări. Pentru descrierea completă a constructorilor, atributelor şi metodelor fiecărei componente aveţi la dispoziţie *javadoc*-ul, care reprezintă documentaţia standard pe care compania Sun o oferă gratuit utilizatorilor. Sugerăm parcurgerea capitolelor într-o manieră încrucişată în funcţie de trimiterile care apar în text, deoarece am folosit deseori „licenţe", prezentând în secvenţele de cod componente care nu au fost descrise la acel moment, din dorinţa de a păstra codul cât mai complet.

13.2.2. Despre interfeţe grafice

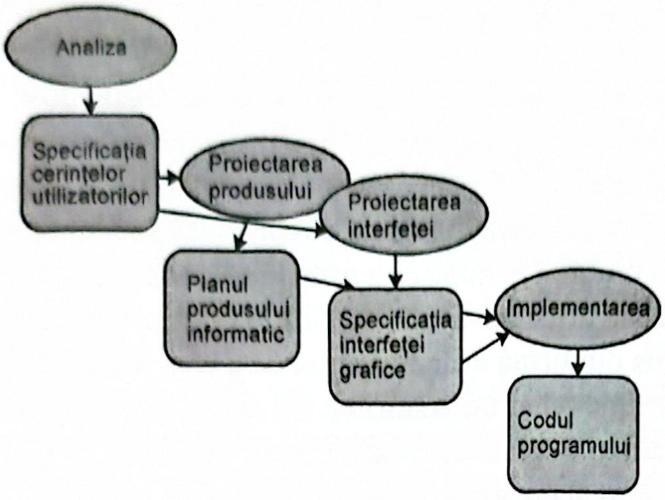
Interfaţa cu utilizatorul, pe scurt UI (engl. *User Interface*), reprezintă totalitatea mecanismelor care permit interacţiunea dintre o aplicaţie şi utilizatorii ei. O particularizare a interfeţei cu utilizatorul o reprezintă interfaţa grafică, pe scurt GUI (engl. *Graphic User Interface*), care se referă strict la comunicarea vizuală dintre utilizator şi aplicaţie. Putem spune că interfaţa grafică este locul de întâlnire dintre utilizatori şi aplicaţia care le este destinată.

Deseori aplicaţiile sunt percepute de utilizatorii finali doar prin intermediul interfeţelor acestora şi, de aceea, o interfaţă realizată deficitar poate compromite întreaga aplicaţie, chiar dacă funcţional ea este bine realizată. În acest sens, o problemă în procesul de realizare a interfeţei grafice o reprezintă posibilitatea ca utilizatorul să nu înţeleagă cum să o utilizeze după ce a fost creată. Acest lucru se întâmplă mai ales pentru că, în general, utilizatorii unei aplicaţii (destinatarii aplicaţiei) nu sunt şi programatori, aceştia din urmă având intenţia de a crea interfaţa conform propriului model mental, diferit de cel al utilizatorilor. De aceea, dezvoltatorii aplicaţiei trebuie să intuiască modalităţile prin care utilizatorii modelează conceptual diferitele funcţionalităţi sau, în general, aspecte ale aplicaţiei, şi să încorporeze aceste modele în viitoarea interfaţă grafică. Metoda folosită în acest proces este analogia concretizată prin metaforă. Acolo unde analogiile sunt greu de realizat, sunt folosite idiomurile care reprezintă convenţii între creatorii şi utilizatorii interfeţei, uşor de reţinut de aceştia din urmă. Spre exemplu, funcţia de ştergere într-o aplicaţie poate fi mai uşor concepută (modelată) de utilizatori ca aruncarea unui obiect nefolositor la coş. Astfel, pentru a şterge o pictogramă, utilizatorul îl trage deasupra coşului de gunoi (engl. *trash*), reprezentat grafic de o altă pictogramă, unde îl eliberează. Acest mod de a vedea lucrurile poartă numele de „înglobarea semanticii în sintaxă", deoarece se încearcă o folosire a simbolurilor care sunt mai uşor de înţeles şi de reţinut. Un alt exemplu ar putea fi folosirea de pictograme în barele de unelte, care ascund printr-o reprezentare grafică uneltele ce se vor folosi după selectarea lor (spre exemplu, o lupă folosită mai apoi ca unealtă pentru realizarea operaţiei de micşorare-mărire). Se poate observa faptul că, în general, există un număr limitat de astfel de metafore şi simboluri care se repetă în mai toate aplicaţiile şi pe toate platformele.

Compania Sun, cea care dă specificaţiile limbajului Java, sugerează folosirea unui set standard de pictograme şi idiomuri, grupate într-o arhivă pe care o găsiţi la adresa *http://developer.java.sun.com/developer/techDocs/hi/repository/*.

Din punctul de vedere al ingineriei programării, momentul în care se pune cu adevărat problema interfeţei cu utilizatorii este cel al procesului de proiectare arhitecturală a aplicaţiei, dar creatorii aplicaţiei vor face aprecieri chiar în procesul de analiză a cerinţelor utilizatorilor. Aşa cum se realizează un plan al produsului informatic, se recomandă realizarea planului interfeţei cu utilizatorul, prin care se clarifică modul cum va arăta interfaţa grafică înainte de a trece la implementarea efectivă a acesteia.

În procesul de proiectare a interfeţelor grafice se poate folosi prototipizarea care presupune realizarea de prototipuri ale interfeţei grafice (eventual, fără funcţionalităţi care se pot adăuga mai apoi), prototipuri care sunt evaluate şi reactualizate, dacă este cazul, pe întreg parcursul dezvoltării aplicaţiei. Prototipurile interfeţei grafice clarifică interacţiunile aplicaţiei cu utilizatorii sau cu alte aplicaţii, respectiv funcţionalităţile aplicaţiei. De asemenea, prin acest procedeu, utilizatorii sunt atraşi în procesul de realizare a aplicaţiei, se pot descoperi unele lipsuri în cerinţele utilizatorilor etc.


Analiza
Specificaţia cerinţelor utilizatorilor
Proiectarea produsului
Proiectarea interfeţei
Planul produsului informatic
Specificaţia interfeţei grafice
Implementarea
Codul programului

Planul interfeţei grafice

Alte informaţii despre principiile care stau la baza proiectării interfeţelor grafice pentru aplicaţiile Java puteţi găsi la adresa *http://java.sun.com/products/jlf/*.

13.2.3. JFC (*Java Foundation Classes*)

JFC reprezintă un grup de API-uri (engl. *Application Programming Interface*) incluse în platforma J2SE (engl. *Java 2 Standard Edition*), care permit dezvoltatorilor să creeze interfeţe grafice (GUI) pentru aplicaţiile Java. *Java Foundation Classes* constă în cinci pachete: AWT, Swing, Accessibility, Java 2D şi Drag and Drop, care se întrepătrund, neavând, în general, o existenţă de sine stătătoare. Java 2D a devenit o parte integrantă a AWT, Swing este construit peste AWT (se aşteaptă ca Swing să se îmbine mult mai puternic cu AWT în următoarele distribuţii Java), suportul Accessibility este construit în Swing, iar API-ul Drag and Drop este integrat în AWT şi Swing. Oferim în tabela care urmează o scurtă prezentare a celor cinci părţi componente ale JFC:

Parte componentă a JFC	Explicaţie
`AWT`	Parte a JFC proiectată pentru a oferi suport pentru crearea interfeţelor grafice, folosind modul grafic. Oferă o ierarhie de clase suficiente pentru proiectarea unei interfeţe grafice nesofisticate pentru un applet sau o aplicaţie. Reprezintă nucleul JFC şi fundamentul pe care s-a construit pachetul Swing.

→

→

`Swing`	Reprezintă un set vast de componente grafice, de la cele mai simple, cum ar fi etichetele sau butoanele, la cele mai complexe ca, de exemplu, tabelele sau arborii. Oferă independență față de platforma peste care este instalată mașina virtuală și posibilitatea personalizării aspectului interfeței și a comportamentului acesteia la acțiunile utilizatorului.
`Accessibility`	Reprezintă posibilitatea de a oferi ajutor utilizatorilor cu deficiențe, cum ar fi posibilitatea de a descifra interfața folosind mesaje vocale, Braille sau dispozitive exterioare atașate calculatorului. Oferă informațiile de care au nevoie programatorii care scriu software pentru astfel de dispozitive auxiliare sau *look-and-feel*-uri. Pachetul `javax.accessibility` este cel care stă la baza acestui sistem, deci cei care scriu programe obișnuite nu au nevoie să-l folosească.
`Java 2D API`	API-ul care permite utilizatorilor să încorporeze în aplicațiile lor grafică 2D complexă, text și imagini.
`Drag and Drop`	Oferă posibilitatea de a realiza operații de *drag-and-drop* între aplicația curentă și aplicațiile native sau între două aplicații Java. La baza acestei facilități stau pachetele `java.awt.dnd` și `java.awt.datatransfer`, dar, în general, nu este nevoie de folosirea lor de către programatori, deoarece sistemul `drag and drop` funcționează implicit.

Diferența majoră dintre Swing și AWT este aceea că, spre deosebire de AWT, componentele Swing sunt scrise 100% în Java, neconținând cod nativ (dependent de sistemul de operare). Aceasta înseamnă că butoanele Swing, spre exemplu, vor arăta și se vor comporta identic pe platforme Macintosh, Solaris, Linux sau Windows. Spre deosebire de cele Swing, butoanele AWT luau mereu aspectul platformei pe care rula aplicația. De asemenea, prin această independență față de platformă se obține o îmbunătățire a vitezei de execuție a aplicațiilor având interfețe realizate în Swing.

API-ul JFC a fost perfecționat și optimizat pe măsura apariției de noi distribuții ale platformei Java și îl regăsim ca parte integrantă în platforma Java 2 Standard Edition. Unele clase și metode s-au învechit (engl. *deprecated*) și s-a renunțat la folosirea lor efectivă pe parcursul apariției noilor distribuții de platforme Java, din cauza ineficienței sau greșelilor pe care le conțineau. Pentru o listă detaliată a acestora, puteți consulta *javadoc*-ul la secțiunea *Deprecated*. Pentru a rula totuși aplicația conținând cod învechit, codul trebuie recompilat folosind opțiunea `-deprecated`.

În particular, biblioteca Swing se poate folosi în aplicații dezvoltate pentru platforme care suportă JFC. Prin intermediul unui *plug-in* pentru browserele Web (purtând numele *Java Plug-in*) puteți să creați interfețe folosind Swing și pentru appleturi Java. *Plug-in*-ul se poate descărca gratuit de la adresa *http://java.sun.com/products/plugin/*. Ca să vizualizați appleturile având interfețe realizate în Swing, fără să folosiți pentru aceasta un browser, aveţi la dispoziţie un utilitar numit `appletviewer` care vine odată cu platforma Java şi pe care îl găsiţi în directorul `/bin` al distribuţiei.

13.2.4. Componentele şi pachetele bibliotecii `Swing`

Componentele `Swing` sunt aproape toate derivate dintr-o singură clasă de bază numită `JComponent`, care moşteneşte, la rândul ei clasa `Container` din AWT. Moştenirea clasei `JComponent` dă componentelor posibilitatea de a avea chenare, *tooltips*, respectiv *look-and-feel* configurabil. De asemenea, tot din `JComponent` se moştenesc şi metodele prin care se pot seta dimensiunile şi poziţionarea componentelor, metode pe care le vom discuta în conţinutul acestei secţiuni. În general, fiecare componentă grafică din AWT are o clasă echivalentă în Swing, al cărei nume se formează din echivalentul AWT la care adăugăm prefixul `J` (singura excepţie este clasa `Canvas`). Reciproca nu este adevărată, deoarece Swing îmbogăţeşte considerabil paleta componentelor disponibile în AWT.

O componentă grafică reprezintă un obiect care are o anumită reprezentare grafică ce poate fi afişată pe ecran şi poate astfel interacţiona cu utilizatorul. Reprezentări grafice au toate obiectele care sunt instanţe ale claselor derivate din `java.awt.Component` şi, în particular, `javax.swing.JComponent` pentru componentele Swing. Pentru a crea un buton având un anumit text drept conţinut, trebuie mai întâi să realizăm o instanţă a clasei `JButton`:

```
JButton buton = new JButton("De acord");
```

Dacă dorim să creăm o listă de cuvinte din care utilizatorul să poată selecta un singur element la un moment dat, trebuie să realizăm mai întâi o instanţă a clasei `JList`, folosind unul dintre constructorii pe care această clasă îi pune la dispoziţie.

```
String continut[]={"AWT", "Swing", "Accessibility", "Java 2D",
  "Drag and Drop"};
JList lista = new JList(continut);
```

În acest caz, butonul şi lista reprezintă componente grafice, iar până aici modul de construcţie a componentelor nu diferă cu nimic de construcţia unui obiect obişnuit. Este de remarcat faptul că crearea obiectelor grafice nu determină şi afişarea reprezentării lor grafice pe ecran. Componentele trebuie să fie aşezate mai întâi pe o suprafaţă de lucru (care mai poartă denumirea de container), la anumite coordonate şi ocupând o anumită zonă a acesteia. Acest lucru se face prin intermediul unui apel al containerului de forma `add(JComponent componenta)`. Trebuie înţeles faptul că nu se pot desena componente fără a fi adăugate iniţial unei suprafeţe de desenare reprezentate printr-un container de bază de tip `JFrame`, `JDialog`, `JWindow` sau `JApplet`.

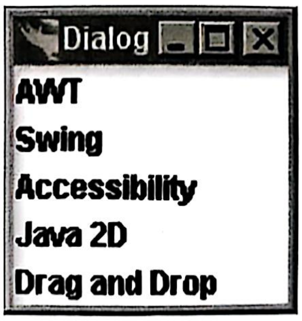
În continuare, vom adăuga o listă unei ferestre `JFrame`, care este una dintre componentele cu rol de container. Pentru mai multe informaţii despre suprafeţele de desenare a se vedea subcapitolul *Containere de bază*.

```
import javax.swing.JFrame;
import javax.swing.JList;
```

```
/** O fereastra */
public class Fereastra extends JFrame {
  // constructorul clasei
  public Fereastra() {
    super("Fereastra");

    // cream o componenta
    String continut[] = { "AWT", "Swing", "Accessibility",
       "Java 2D", "Drag and Drop" };
    JList lista = new JList(continut);

    // adaugam componenta
    this.getContentPane().add(lista);
  }
}
```

După aşezarea în container, componentele grafice vor deveni vizibile doar odată cu vizualizarea întregii suprafeţe de desenare printr-un apel `setVisible(true)`.

```
/** Aplicatia pentru fereastra*/
public class Aplicatia {
  public static void main(String[] args) {
    // cream fereastra
    Fereastra fereastra = new Fereastra();

    // facem vizibila fereastra
    fereastra.pack();
    fereastra.setVisible(true);
  }
}
```



AWT
Swing
Accessibility
Java 2D
Drag and Drop

Fereastra aplicaţiei

De asemenea, chiar dacă vizualizarea containerului a fost deja realizată, prin adăugarea unei noi componente containerului, aceasta va deveni, la rândul ei, vizibilă fără a mai fi nevoie de un apel `setVisible()`. Folosind un apel de forma `setVisible(false)` aplicat unei componente deja vizualizate, aceasta se va ascunde. Dacă este un container conţinând şi alte componente, acestea vor fi la rândul lor ascunse.

Modul în care componentele unui container vor fi desenate la momentul vizualizării acestuia depinde de gestionarul de poziţionare care este asociat containerului. Acesta determină cum vor fi poziţionate componentele pe suprafaţa de desenare şi ce dimensiuni vor avea. Gestionarii sunt instanţe ale claselor care implementează interfaţa `LayoutManager`, cum ar fi: `BorderLayout`, `BoxLayout`, `CardLayout`, `FlowLayout`, `GridLayout`, `GridBagLayout` sau `SpringLayout`. Interfaţa `LayoutManager2` este derivată din `LayoutManager` şi este extinsă de cei mai mulţi dintre gestionarii enumeraţi mai sus, care oferă acestora suport pentru aliniamentul componentelor conţinute şi gestiunea dimensiunilor maxime.

Prezentăm în continuare modalitatea prin care se ataşează un gestionar unui container. Spre exemplu, deoarece gestionarul `BorderLayout`, care este asociat implicit containerului ce ţine conţinutul unei ferestre, nu respectă dimensiunile asociate componentelor de către dezvoltator, îi vom asocia un alt gestionar de poziţionare folosind metoda `setLayout()`.

```
JPanel continut = this.getContentPane();
continut.setLayout(new FlowLayout());
```

Când componenta nu este gestionată de un gestionar de poziţionare (acest lucru se realizează printr-un apel `setLayoutManager(null)` pentru container), atunci ea va fi întotdeauna afişată cu dimensiunile şi poziţia care i-au fost atribuite în prealabil. Dacă nu i s-au asociat dimensiuni, implicit acestea vor fi `0`, iar dacă nu a fost poziţionată, va fi aşezată în colţul din stânga-sus al containerului.

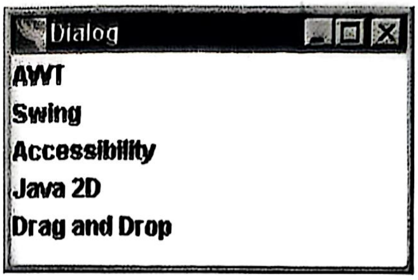
Este esenţială deci înţelegerea modului în care lucrează gestionarii de poziţionare, pentru a nu avea surprize la momentul desenării containerului. De asemenea, trebuie ştiut că, implicit, panourilor reprezentând containere intermediare li se asociază managerul `FlowLayout`, iar panoului conţinut al unui container de bază i se asociază gestionarul `BorderLayout`. Se poate alege ca mai multe componente să fie grupate folosind containere intermediare (a se vedea secţiunea *Containere intermediare*), astfel având posibilitatea de a le gestiona împreună poziţionarea, separat de toate celelalte componente.

Fiecare componentă grafică are asociate trei dimensiuni (de fapt, proprietăţi) explicate în tabelul care urmează împreună cu metodele prin care acestea pot fi accesate:

Dimensiuni	Metode de acces	Explicaţii
`preferredSize`	**`getPreferredSize()`** **`setPreferredSize()`**	Dimensiunea preferată a unei componente. Este folosită de cei mai mulţi gestionari de poziţionare pentru a dimensiona implicit componentele. Dimensiunea preferată a unei componente este determinată de *look-and-feel*-ul componentei şi de fonturile pe care componenta le foloseşte.
`minimumSize`	**`getMinimumSize()`** **`setMinimumSize()`**	Folosită de gestionarii de poziţionare ca limită în micşorarea dimensiunilor componentelor.
`maximumSize`	**`getMaximumSize()`** **`setMaximumSize()`**	Folosită de gestionarii de poziţionare ca limită în maximizarea dimensiunilor componentelor.

Metodele de forma `setXXXSize()`, prezentate în tabelul precedent, primesc ca intrare o instanţă a clasei `java.awt.Dimension`. În exemplul următor, vom seta dimensiunea listei din exemplul precedent pentru a avea lăţimea de 200 pixeli şi înălţimea de 100 pixeli.

```
Dimension dim = new Dimension(200,100);
lista.setPreferredSize(dim);
```



AWT
Swing
Accessibility
Java 2D
Drag and Drop

Fereastra mărită

Pentru ca o componentă să fie desenată într-adevăr cu dimensiunile pe care le-am specificat în prealabil, trebuie să ne asigurăm că aceasta este administrată de un gestionar de poziţionare care respectă aceste setări. Spre exemplu, `FlowLayout` şi `GridBagLayout` respectă dimensiunile preferate ale componentelor. Nu acelaşi lucru se întâmplă cu `BorderLayout` şi `GridLayout`. Gestionarul `BoxLayout` este singurul gestionar care respectă dimensiunile maxime. Dacă se doreşte redimensionarea unei componente care este deja vizibilă, ea trebuie redesenată după setarea noilor dimensiuni. Pentru a aduce componentele la dimensiunile lor preferate (engl. *preferred size*), vom apela metoda `pack()` pentru container înainte de a apela `setVisible()`.

Pe lângă setarea dimensiunilor putem, de asemenea, modifica poziţia fiecărei componente în interiorul containerului. Când componenta nu este gestionată de un manager de poziţionare, acest lucru este uşor. În schimb, un manager de poziţionare poate neglija dimensiunile pe care le dăm componentei. Pentru mai multe amănunte puteţi consulta secţiunea dedicată gestionarilor de poziţionare din capitolul dedicat appleturilor şi componentelor AWT.

Metoda `setBounds()` moştenită din `JComponent` poate fi folosită pentru a seta în acelaşi timp mărimea şi poziţionarea unei componente în interiorul containerului-părinte. Prezentăm în continuare cele două supraîncărcări ale metodei folosite în cazul exemplului anterior:

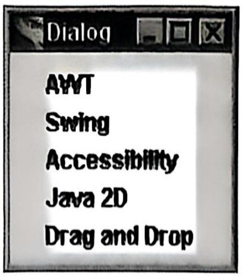
```
lista.setBounds(200, 200, 400, 300);
```

sau

```
Rectangle drept = new Rectangle(200, 200, 400, 300);
lista.setBounds(drept);
```

Pentru a obţine efectul scontat, va trebui ca după apelarea constructorului din superclasă să indicăm folosirea gestionarului pentru poziţii absolute, astfel:

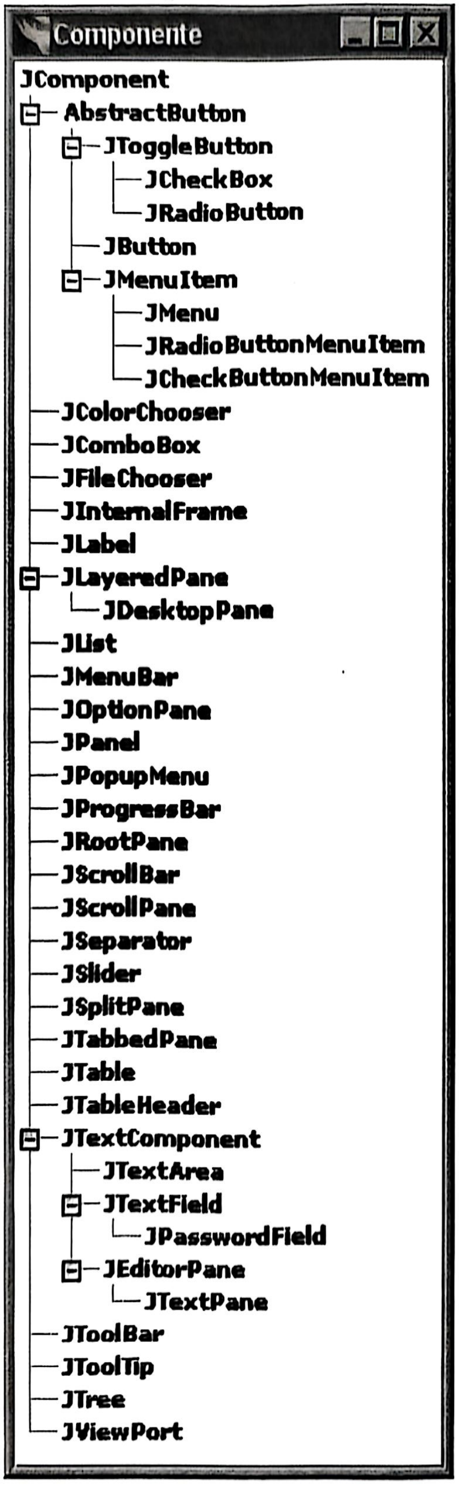
```
this.setLayout(null);
```



AWT
Swing
Accessibility
Java 2D
Drag and Drop

Fereastra cu poziţionare absolută

Prezentăm în continuare ierarhia parţială a componentelor Swing:



- JComponent
  - AbstractButton
    - JToggleButton
      - JCheckBox
      - JRadioButton
    - JButton
    - JMenuItem
      - JMenu
      - JRadioButtonMenuItem
      - JCheckButtonMenuItem
  - JColorChooser
  - JComboBox
  - JFileChooser
  - JInternalFrame
  - JLabel
  - JLayeredPane
    - JDesktopPane
  - JList
  - JMenuBar
  - JOptionPane
  - JPanel
  - JPopupMenu
  - JProgressBar
  - JRootPane
  - JScrollBar
  - JScrollPane
  - JSeparator
  - JSlider
  - JSplitPane
  - JTabbedPane
  - JTable
  - JTableHeader
  - JTextComponent
    - JTextArea
    - JTextField
      - JPasswordField
    - JEditorPane
      - JTextPane
  - JToolBar
  - JToolTip
  - JTree
  - JViewPort

Ierarhia parţială a componentelor Swing

De asemenea, toate componentele grafice Swing sunt construite conform specificaţiei JavaBeans, ceea ce înseamnă că sunt, de fapt, componente JavaBeans. Mai multe amănunte despre acest aspect găsiţi în secţiunea „Alinierea la JavaBeans". Ce putem spune acum este că această compatibilitate dă posibilitatea construirii de medii vizuale pentru Java în care interfaţa grafică pentru aplicaţii se construieşte interactiv prin aşezarea componentelor grafice pe suprafaţa de lucru. Astfel, se pot testa efectiv mai multe posibilităţi de a proiecta interfaţa aplicaţiei până se ajunge la o formă definitiv acceptată.

Pentru a putea folosi componentele Swing trebuie să importăm mai întâi pachetele corespunzătoare. În general, este de ajuns să includem următoarele pachete:

- `javax.swing`
  pentru componente Swing simple;
- `javax.swing.event`
  pentru evenimentele generate de componentele Swing;
- `java.awt`
  pentru componentele din AWT;
- `java.awt.event`
  pentru evenimente generale definite în AWT.

Prezentăm lista completă a pachetelor Swing care trebuie incluse în aplicaţie în funcţie de necesităţi.

Pachet	Prezentare
`javax.swing`	Conţine componentele Swing de bază şi interfeţele implicite pentru componentele care dispun de ele. Pentru componentele complexe se definesc pachete separate.
`javax.swing.border`	Clase şi interfeţe folosite pentru a defini diferite stiluri de chenare (engl. *borders*). De remarcat faptul că acestea nu sunt componente grafice.
`javax.swing.colorchooser`	Clasele şi interfeţele care oferă suport pentru componenta `JColorChooser` ce permite selectarea de culori dintr-o paletă.
`javax.swing.event`	Pachetul care conţine evenimentele proprii Swing (engl. *event*), respectiv ascultătorii (engl. *listener*). Componentele Swing suportă, de asemenea, evenimente diferite în pachetele `java.awt.event` şi `java.beans`.
`javax.swing.filechooser`	Clasele şi interfeţele ce oferă suport pentru componenta `JFileChooser`, cea cu ajutorul căreia se poate face selecţia fişierelor.

➔

➔

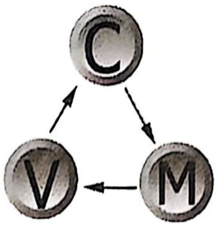
`javax.swing.plaf`	Conţine API-ul pentru *look-and-feel*-ul prin care utilizatorii pot personaliza interfaţa componentelor. Cele mai multe clase din acest pachet sunt abstracte, implementatorii trebuind să le suprascrie pentru a crea diverse stiluri aplicabile interfeţelor grafice.
`javax.swing.plaf.basic`	Conţine clasele şi interfeţele pe baza cărora se construiesc pachetele *look-and-feel* efective. Clasele din acest pachet trebuie suprascrise pentru a crea propriul pachet *look-and-feel*.
`javax.swing.plaf.metal`	*Metal* este *look-and-feel*-ul implicit al componentelor Swing.
`javax.swing.plaf.multi`	Acest pachet nu reprezintă un *look-and-feel*, ci oferă posibilitatea combinării mai multor *look-and-feel*-uri deja existente, putându-se astfel folosi concomitent. Spre exemplu, se poate combina un *look-and-feel* audio cu unul vizual.
`javax.swing.table`	Clase şi interfeţe care oferă suport pentru componenta `JTable`, care este folosită pentru reprezentarea datelor sub formă de tabele. Permite, de asemenea, personalizarea parţială a modului de vizualizare a tabelei fără ajutorul sistemului *look-and-feel*.
`javax.swing.text`	Clase şi interfeţe folosite de componente text oferind suport pentru diverse aspecte ale acestora.
`javax.swing.text.html`	Suport pentru componentele text care vizualizează HTML.
`javax.swing.text.html.parser`	Suport pentru procesarea HTML.
`javax.swing.text.rtf`	Suport pentru documente RTF.
`javax.swing.tree`	Clase şi interfeţe care oferă suport pentru componenta `JTree`, care este folosită pentru reprezentarea datelor sub formă de arbore. Permite, de asemenea, personalizarea parţială a modului de vizualizare a arborelui fără ajutorul sistemului *look-and-feel*.
`javax.swing.undo`	Pachet care oferă suport pentru implementarea şi gestionarea funcţionalităţii *undo/redo*.

13.2.5. Principii de bază

Această secţiune descrie aspecte legate de modul în care a fost construită biblioteca Swing. Puteţi începe direct cu următoarele secţiuni, care descriu componentele, şi puteţi reveni aici în cazul în care doriţi să aprofundaţi. Pentru a înţelege cum funcţionează componentele mai complexe, precum `JTree` sau `JTable`, este necesar studiul acestei secţiuni.

13.2.5.1. *MVC (Model-View-Controller)*

MVC este filosofia (nivelul conceptual) clasică pentru proiectarea interfeţelor grafice în limbajele obiectuale care datează din anii 1970. Reprezintă structurarea componentelor în trei părţi: un model, o vizualizare (engl. *view*) şi un controler (engl. *controller*), fiecare având propria utilitate şi funcţionalitate şi interacţionând cu celelalte părţi aşa cum se poate vedea în figura care urmează:



Interacţiunile din MVC

Componenta devine un mediator între model, vizualizare şi controler. Unul dintre avantajele acestei modalităţi de a structura componentele este acela că ne permite să personalizăm *look-and-feel*-ul unei componente fără a modifica modelul. Putem să asociem unui acelaşi model (deci aceloraşi date) mai multe moduri de vizualizare chiar în acelaşi timp.

Rolurile părţilor unei componente sunt explicate în tabelul prezentat mai jos.

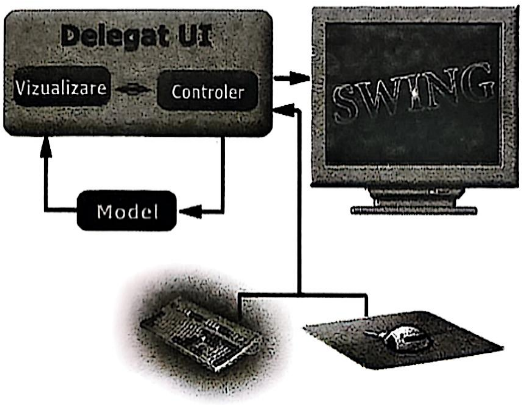
Partea componentă	Explicaţie
`Model`	Modelul păstrează, în general, datele cărora se doreşte să li se dea o vizualizare numindu-se, în acest caz, model de date (engl. *data model*). Spre exemplu, pentru o componentă text, datele reprezintă textul ce se vrea vizualizat. Modelele comunică indirect cu vizualizarea şi controlerul prin intermediul sistemului de evenimente şi nu deţin referinţe la nici una din acestea. Pot să existe modele care să nu ţină neaparat datele componentei, ci stări ale vizualizării grafice pentru componentă (engl. *GUI-state*). Un exemplu în acest sens ar putea fi un model pentru o listă care încapsulează modulul de selecţie sau pentru un buton, un model care încapsulează starea butonului: armat, apăsat etc. În cazul acestui ultim tip de modele, componentele dispun de metode pentru manipularea stării lor fără a face apel la modele. Trebuie precizat că deseori modelele încorporează atât datele componentei, cât şi stări ale vizualizării componentelor.

→

`Vizualizare`	Reprezentarea vizuală a modelului descriind pentru componentă modul cum arată (engl. *look*). Vizualizarea este responsabilă pentru păstrarea pe ecran a reprezentării actualizate, primind în acest scop mesaje de la model şi controler.
`Controler`	Controlerul ia datele de intrare ale utilizatorilor din vizualizare şi le transformă în schimbări corespunzătoare în model. Controlerul are rolul de a determina când şi cum va reacţiona componenta la evenimentele de intrare dinspre dispozitivele de intrare cum ar fi mouse-ul şi tastatura. În acest sens reprezintă comportamentul (engl. *feel*) componentei.

13.2.5.2. UI Delegate

Componentele Swing se bazează pe o versiune mai modernă a modelului de structurare MVC. Aceasta poartă denumirea de *model-delegat* şi se deosebeşte de *model-view-controller* prin faptul că vizualizarea şi controlerul componentei sunt compactate într-o singură clasă care poartă denumirea de *UI delegate*. Această unire are loc deoarece este foarte greu să se scrie un controler generic care să nu ştie nimic despre vizualizarea pe care o ascultă. Figura următoare explică acest model:



Funcţionarea versiunii *model-delegat*

Pentru fiecare componentă Swing există câte o clasă *UI delegate* derivată din clasa abstractă `ComponentUI`, prin care se gestionează *look-and-feel*-ul componentei. Numele acestei clase se formează de la numele componentei de la care se înlătură prefixul `"J"` şi se adaugă sufixul `"UI"`. Toate clasele delegat se află în pachetul `javax.swing.plaf`.

Metoda având prototipul `protected void setUI(ComponentUI newUI)`, definită în clasa `JComponent`, permite asocierea unui delegat unei componente.

Metoda având prototipul `static ComponentUI CreateUI(JComponent c)` este folosită pentru a obţine o instanţă a clasei delegat pentru componenta `c`.

Folosind aceste două metode putem, spre exemplu, să schimbăm *look-and-feel*-ul unei componente de tip `JButton` în felul următor:

```
JButton sterge=new JButton();
sterge.setUI(
   (KunststoffButtonUI)KunststoffButtonUI.createUI(sterge));
```

13.2.5.3. Look-and-feel

Swing permite dezvoltatorilor crearea de pachete care să conţină clasele delegat pentru mai multe componente sau toate tipurile componentelor pe care Swing le pune la dispoziţie. Aceste pachete se numesc implementări *pluggable look-and-feel* (PLAF) şi cu ajutorul lor se poate schimba stilul interfeţei grafice chiar în momentul în care aceasta rulează (engl. *run-time*).

Pachetul `javax.swing.plaf` conţine clasele abstracte (corespunzătoare componentelor clasice Swing), care extind clasa `ComponentUI`, acestea fiind la rândul lor extinse în pachetul `javax.swing.plaf.basic`. Acest ultim pachet nu este un *look-and-feel* utilizabil, ci stă la baza implementărilor efective ale dezvoltatorilor de *look-and-feel*.

Există trei implementări care au la bază *Basic look-and-feel*, care vin odată cu distribuţiile de platforme Java şi se află în pachetele:

- Windows: `com.sun.java.swing.plaf.windows.Windowslook-and-feel`
- CDE\Motif: `com.sun.java.swing.plaf.motif.Motiflook-and-feel`
- Metal (*default*): `javax.swing.plaf.metal.Metallook-and-feel`

Java 2 pune la dispoziţia posesorilor de sisteme Macintosh pachetul care simulează interfaţa-utilizator Macintosh, dar acesta trebuie descărcat separat de pe reţea. Metal este *look-and-feel*-ul implicit al interfeţelor, iar Windows şi Macintosh sunt suportate doar pe platformele corespunzătoare.

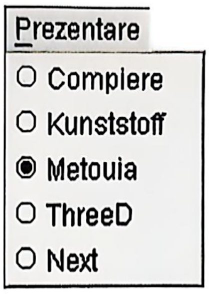
De asemenea, există deja o serie de implementări realizate de terţe părţi. O listă a lor puteţi găsi la adresa: *http://www.javootoo.com*, de unde le puteţi şi descărca. Sunteţi încurajaţi în scrierea de plug-in-uri pentru interfeţele voastre grafice, având în acest sens la dispoziţie un ghid preţios *Java Look-and-Feel Design Guidelines* despre care puteţi afla mai multe la adresa *http://java.sun.com/products/jlf/at/book/index.html*.

Prezentăm două vizualizări ale unei aceleiaşi ferestre de dialog, dar folosind stiluri diferite. Primul, *Metal look-and-feel*, iar cel de al doilea, *Kunststoff*, pe care îl puteţi descărca gratuit de la adresa prezentată mai sus. A se vedea nuanţele de gri din a doua fereastră.

Pentru a putea schimba stilul aplicaţiei, chiar şi în timpul rulării, trebuie să căutaţi în pachetul ce conţine *plug-in*-ul o clasă derivată din `javax.swing.LookAndFeel`, al cărei nume este format din rădăcina `LookAndFeel` la care se adaugă prefixul ce dă numele *plug-in*-ului (de exemplu, `MetalLookAndFeel`). Această clasă face legătura între *plug-in* şi aplicaţie. De asemenea, setarea *plug-in*-ului curent se face prin intermediul clasei `javax.swing.UIManager`, după cum putem vedea în exemplul următor prin intermediul căruia se setează ca *plug-in* curent *Motif*:

```
try {
   UIManager.setLookAndFeel(
      "com.sun.java.swing.plaf.motif.MotifLookAndFeel");
   // am setat look-and-feel-ul Motif interfetei grafice
   SwingUtilities.updateComponentTreeUI(this);
   // aceasta metoda este folosita pentru a imprima acelasi
   // look-and-feel si
   // componentelor continute in containerul curent
}
catch (Exception  e) {
 System.err.println("Nu se poate utiliza acest look-and-feel");
}
```

Prezentăm în continuare o clasă pentru gestionarea *plug-in*-urilor prin încărcarea dinamică a claselor de apelare:



Meniu pentru schimbarea *look-and-feel*-ului

```
import java.awt.event.*;
import javax.swing.*;

/**
 * Clasa ofera un meniu pentru selectarea
 * look-and-feel-ului pentru o fereastra
 */
public class LF extends JMenu implements ActionListener {
   /** Referinta la fereastra careia i se va aplica
    * look-and-feel-ul. */
   JFrame frame;

   /** Numele look-and-feel-urilor disponibile */
   String[] lf = { "Compiere",
         "Kunststoff",
         "Metouia",
         "ThreeD",
         "Next"
   };
z
```

```
/** Pachetele din care fac parte look-and-feel-urile
 * date de <tt>lf</tt>
 */
String[] pachete = {
    "org.compiere.plaf.",
    "com.incors.plaf.kunststoff.",
    "net.sourceforge.mlf.metouia.",
    "swing.addon.plaf.threeD.",
    "nextlf.plaf."
};

/**
 * Constructorul clasei
 * @param name - numele meniului
 * @param frame - fereastra pentru care se va aplica
 *                look-and-feel-ul selectat.
 */
public LF(String name, JFrame frame) {
  super(name);
  this.frame = frame;

  // utilizam un grup de butoane pentru selectare
  ButtonGroup group = new ButtonGroup();
  for (int i = 0; i < lf.length; i++) {
    JRadioButtonMenuItem optiune =
        new JRadioButtonMenuItem(lf[i]);
    // adaugam optiunea curenta grupului de butoane
    group.add(optiune);
    // adaugam optiunea curenta meniului
    add(optiune);
    optiune.setActionCommand(i + "");
    // adaugam fiecarei optiuni ascultatorul
    optiune.addActionListener(this);
  }
}

/**
 * Metoda apelata la aparitia evenimentului
 * de selectare a unei optiuni
 * @param e evenimentul care a aparut
 */
public void actionPerformed(ActionEvent e) {
  try {
    // detectam numarul optiunii selectate
```

```
      int poz = Integer.parseInt(e.getActionCommand());
      /* construim numele complet al clasei
       * corespunzatoare look-and-feel-ului selectat */
      String nume = pachete[poz] + lf[poz] + "LookAndFeel";
      // folosind Reflection API incarcam clasa
      Class clasa = Class.forName(nume);
      // cream o instanta a clasei
      LookAndFeel l = (LookAndFeel)clasa.newInstance();
      // setam look-and-feel-ul aplicatiei
      UIManager.setLookAndFeel(l);
      // actualizam look-and-feel-ul ferestrei-tinta
      SwingUtilities.updateComponentTreeUI(frame);
    } catch (Exception ex) {
      System.err.println("Eroare la incarcat l&f! " + ex);
    }
  }
}
```

Folosind clasa descrisă mai sus puteţi da utilizatorului aplicaţiei posibilitatea să-şi schimbe stilul interfeţei grafice prin intermediul unui meniu cu butoane radio, chiar în timpul rulării programului:

```
JMenu meniuTeme = new LF("Prezentare", this);
meniuBara.add(meniuTeme);
```

13.2.5.4. *Mai multe despre modele*

Un alt avantaj pe care sistemul MVC îl oferă este posibilitatea de a personaliza modelul de date, acest lucru fiind posibil datorită despărţirii lui de vizualizare. Spre exemplu, putem crea zone de text (engl. *text area*) care să accepte numai introducerea de cifre, un anumit număr de caractere sau să nu reacţioneze la apăsarea unor taste cum ar fi *Enter* sau *Tab* (a se vedea secţiunea dedicată componentelor text).

Prezentăm lista interfeţelor care reprezintă modele şi corespunzător componentele care utilizează, respectiv structurile de date pe care le încapsulează:

Interfaţa model	Componentele care folosesc acest model	Structura de date pe care o înfăşoară
BoundedRangeModel	JProgressBar, JScrollBar, JSlider	Patru întregi: value, extent, min, max, cu următoarele restricţii: value şi extent trebuie sa fie iniţializate înainte de a specifica valorile min şi max; extent este întotdeauna <= max şi >= value.

➔

➔

`ButtonModel`	Toate subclasele componentei `AbstractButton`	O valoare booleană reprezentând starea în care se află butonul (dacă butonul este selectat sau nu).
`ListModel`	`JList`	O colecție de obiecte.
`ComboBoxModel`	`JComboBox`	O colecție de obiecte și un obiect selectat.
`MutableComboBoxModel`	`JComboBox`	Un `Vector` sau o altă colecție de obiecte și un obiect selectat.
`ListSelectionModel`	`JList`, `TableColumnModel`	Unul sau mai mulți indici corespunzând elementelor selectate dintr-un tablou sau listă. Permite selectatea unui anumit element, interval sau a mai multor intervale.
`SingleSelectionModel`	`JMeniuBar`, `JPopupMenu`, `JMenuItem`, `JTabbedPane`	Indexul elementului selectat dintr-o colecție gestionată de cel care o implementează.
`ColorSelectionModel`	`JColorChooser`	O instanța a clasei `Color` reprezentând o culoare.
`TableModel`	`JTable`	O matrice bidimensională de obiecte.
`TableColumnModel`	`JTable`	O colecție de obiecte `TableColumn`, un set de ascultători pentru evenimentele legate de coloane, distanța dintre coloane, un model pentru selecție și un indicator pentru coloane selectate.
`TreeModel`	`JTree`	Structuri arborescente. Implementările trebuie să fie capabile să distingă nodurile frunză de cele interioare.
`TreeSelectionModel`	`JTree`	Nodurile selectate. Sunt permise selecții continue și discontinue.
`Document`	Toate componentele text	Conținut sub formă de text. Implementările mai complexe suportă documente formatate.

Componentele Swing care au modele permit obținerea unei referințe spre acestea sau adăugarea unui model nou prin intermediul metodelor `getModel()`, respectiv `setModel()`. Dacă nu se setează un model în momentul construcției componentei, un model implicit este creat și instalat intern în componentă. Prin convenție, acesta are un nume format din prefixul `"Default"` adăugat numelui modelului. Pentru componentele mai complexe, sunt puse la dispoziția programatorilor modele abstracte pentru care aceștia să nu fie nevoiți să creeze modele pornind de la nimic. Numele acestora se formează din prefixul `"Abstract"` adăugat numelui modelului. A se vedea secţiunea dedicată componentei `JTable` pentru exemplificare.

Modelele nu deţin informaţii despre vizualizările componentei în spatele cărora stau. Această abordare este necesară pentru a permite mai multe vizualizări pentru un acelaşi model. Un model nu deţine decât o mulţime de ascultători ataşaţi şi interesaţi în a şti când se modifică starea modelului. Componenta este cea care prin intermediul ascultătorilor modelului îşi modifică vizualizarea, redesenându-se.

Evenimentele generate de modele în momentul schimbării stării pot fi de două feluri. Pot fi evenimente „uşoare", care nu poartă cu ele informaţii despre modificările survenite în model, ci se aşteaptă ca ascultătorii să trimită un mesaj modelului şi să afle ce s-a modificat. Aceste tipuri de evenimente sunt folosite în cazul în care starea componentei se schimbă des şi sunt prezentate în tabelul următor:

Model	Ascultător	Eveniment
`BoundedRangeModel`	`ChangeListener`	`ChangeEvent`
`ButtonModel`	`ChangeListener`	`ChangeEvent`
`SingleSelectionModel`	`ChangeListener`	`ChangeEvent`

Singura informaţie pe care un astfel de eveniment o conţine este sursa care l-a generat. Prezentăm un exemplu de tratare a unui astfel de eveniment:

```
JSlider slider = new JSlider();
BoundedRangeModel model = slider.getModel();
model.addChangeListener(new ChangeListener() {
   public void stateChanged(ChangeEvent e) {
      BoundedRangeModel m = (BoundedRangeModel)e.getSource();
      // am obtinut o referinta la generatorul evenimentului
      System.out.println("Modelul este " + m.toString());
}});
```

Alte tipuri de evenimente sunt cele care descriu precis ceea ce s-a întâmplat în model atunci când au fost generate. O listă a acestora o prezentăm în continuare:

Model	Ascultător	Eveniment
`ListModel`	`ListDataListener`	`ListDataEvent`
`ListSelectionModel`	`ListSelectionListener`	`ListSelectionEvent`
`ComboBoxModel`	`ListDataListener`	`ListDataEvent`
`TreeModel`	`TreeModelListener`	`TreeModelEvent`
`TreeSelectionModel`	`TreeSelectionListener`	`TreeSelectionEvent`
`TableModel`	`TableModelListener`	`TableModelEvent`
`TableColumnModel`	`TableColumnModelListener`	`TableColumnModelEvent`
`Document`	`DocumentListener`	`DocumentEvent`
`Document`	`UndoableEditListener`	`UndoableEditEvent`

Prezentăm un exemplu de folosire a unui astfel de ascultător în cazul modelului `ListSelectionModel`, în care afişăm indexul elementului selectat, obţinut din evenimentul generat:

```
String elemente[] = {"Unu", "Doi", "Trei"};
JList lista = new JList(elemente);
ListSelectionModel model = lista.getSelectionModel();
model.addListSelectionListener (new ListSelectionListener() {
   public void valueChanged(ListSelectionEvent e) {
      // obtinem informatii de la evenimentul aruncat
      if (!e.getValueIsAdjusting()) {
         System.out.println("S-a schimbat selectia: "
         +e.getFirstIndex());
      }
   }});
```

Nu toate componentele au modele. În general, cele care au funcţionalitate de container, cum ar fi `JFrame`, `JDektopPane`, nu dispun de modele. La fel, componentele simple sau complexe în spatele cărora nu sunt multe informaţii. Au în schimb cele interactive, cum ar fi `JButton`, `JTextField`, `JTable`. De asemenea, unele componente au mai multe modele, fiecare dintre ele participând la o parte din modelarea componentei. Spre exemplu, `JTable` are un model pentru a păstra informaţii referitoare la selectare şi un model pentru a păstra datele. Un acelaşi model poate fi folosit de mai multe componente, aşa cum se întâmplă în cazul modelului `BoundedRangeModel` folosit atât de `JSlider`, cât şi de `JScrollbar`. După cum se observă, separarea modelului nu este o regulă atât de des întâlnită în Swing cum este separarea vizualizării şi comportamentului.

13.2.5.5. *Swing* versus *AWT*

Componentele din AWT sunt numite componente „grele" (engl. *heavyweights*), în timp ce acelea din Swing (în afară de containerele derivate direct din clasele AWT corespondente) sunt numite componente „uşoare" (engl. *lightweights*). Desenarea componentelor grele se face prin intermediul unor componente asociate (engl. *peer*) sistemului de operare, fiind astfel dependentă de sistemul peste care este instalată platforma Java. Aceste componente *peer* dau şi *look-and-feel*-ul componentelor corespunzătoare din AWT şi de aceea interfeţele realizate în AWT au întotdeauna *look-and-feel*-ul sistemului de operare. În Swing, prin independenţa faţă de aceste componente native peer, componentele îşi gestionează singure l*ook-and-feel*-ul (sau, mai bine spus, dau utilizatorului această posibilitate), făcând astfel ca modul cum arată şi cum se comportă o interfaţă grafică să fie identic, indiferent de sistemul de operare sau maşina pe care rulează aplicaţia. Şi, în plus, dau posibilitatea de a schimba *look-and-feel*-ul în mod dinamic chiar în timpul rulării aplicaţiei.

Din cauza acestor diferenţe între aplicaţii, atunci când adăugăm o componentă grea într-un container, aceasta va acoperi toate componentele uşoare deja existente acolo. De asemenea, componentele grele nu permit transparenţa. De aceea, trebuie să avem grijă atunci când utilizăm împreună într-o aceeaşi aplicaţie componente AWT şi Swing. Clasa `java.awt.Toolkit` face legătura între componentele „grele", cum ar fi cele AWT, şi componentele corespondente *peer* specifice sistemului de operare pe care este instalată platforma Java. Această clasă reprezintă fundamentul pe care este construit pachetul AWT. În aplicaţii vom folosi această clasă doar pentru a obţine rezoluţia cu ajutorul căreia vom realiza poziţionarea absolută a containerelor de bază.

13.3. Fundamentele Swing

13.3.1. Evenimente şi ascultători

În Swing, interfaţa grafică (reprezentată de componentele grafice membre) este separată clar de implementare, care reprezintă codul ce se doreşte a fi executat în urma interacţiunii utilizatorului cu interfaţa. Din acest motiv, prin construcţie, Swing a fost conceput ca un sistem condus de evenimente. Evenimentele sunt generate (engl. *fire*) de fiecare dată când apăsăm o tastă sau un buton al mouse-ului şi sunt tratate corespunzător prin cod scris de programator. Acest mod în care componentele aruncă evenimente şi felul în care acestea sunt tratate a rămas neschimbat de la platforma JDK 1.1 în care a fost implementat pentru prima oară.

Sunt multe tipuri de evenimente pe care componentele Swing le pot genera. Fiecare eveniment este înfăşurat într-un obiect derivat din clasa `java.util.EventObject` prin intermediul căruia programatorul mânuieşte evenimentul, putând determina sursa sa (componenta care îl generează) şi, uneori, informaţii despre tipul evenimentului, starea sursei înainte şi după ce evenimentul a fost generat. Sursele evenimentelor sunt, în general, componente sau modele, dar există şi alte tipuri de obiecte care generează evenimente. În cazul evenimentelor derivate din clasa `EventObject`, sursa se determină efectiv folosind metoda `public Object getSource()`. Informaţii despre tipul evenimentului se obţin printr-un apel al metodei `publicString toString()`. Evenimentele sunt grupate în două pachete: clasicul `java.awt.event`, care se referă la evenimente generale, respectiv `java.swing.event`, care se referă la evenimentele specifice componentelor grafice Swing.

Pentru a trata evenimentul generat de o componentă, programatorul trebuie să asocieze acesteia un obiect ascultător (engl. *listener*) care conţine descrierea reacţiei aplicaţiei la apariţia evenimentului respectiv. Dacă un eveniment are tipul `XXEvent`, atunci ascultătorul este o implementare a interfeţei `XXListener`, care se găseşte, de asemenea, în unul dintre pachetele `java.awt.event` sau `java.swing.event`.

Prezentăm lista evenimentelor de bază din Swing, a ascultătorilor, a metodelor de adăugare, ştergere, respectiv componentele pentru care se aplică. Desigur există şi alte evenimente derivate din cele de bază, specifice anumitor componente.

Evenimente, ascultători şi modele de adăugare-ştergere	Componentele care suportă aceste evenimente

`ActionEvent`
`ActionListener`
`addActionListener()`
`removeActionListener()`
	`JButton`, `JList`, `JTextField`, `JMenuItem` şi componentele derivate, inclusiv `JCheckBoxMenuItem`, `JMenu` şi `PopupMenu`

`AdjustmentEvent`
`AdjustmentListener`
`addAdjustmentListener()`
`removeAdjustmentListener()`
	`JScrollBar` şi orice implementare a interfeţei `Adjustable`

`ComponentEvent`
`ComponentListener`
`addComponentListener()`
`removeComponentListener()`
	`Component` şi derivatele incluzând `JButton`, `JCheckBox`, `JComboBox`, `Container`, `JPanel`, `JApplet`, `JScrollPane`, `Window`, `JDialog`, `JFileDialog`, `JFrame`, `JLabel`, `JList`, `JScrollBar`, `JTextArea` şi `JTextField`

`ContainerEvent`
`ContainerListener`
`addContainerListener()`
`removeContainerListener()`
	`Container` şi derivatele, incluzând `JPanel`, `JApplet`, `JScrollPane`, `Window`, `JDialog`, `JFileDialog` şi `JFrame`

`FocusEvent`
`FocusListener`
`addFocusListener()`
`removeFocusListener()`
	`Component` şi derivate

`KeyEvent`
`KeyListener`
`addKeyListener()`
`removeKeyListener()`
	`Component` şi derivate

`MouseEvent` (folosit şi pentru clic, şi pentru mişcare)
`MouseListener`
`addMouseListener()`
`removeMouseListener()`
	`Component` şi derivate

`MouseEvent` (folosit şi pentru clic, şi pentru mişcare)
`MouseMotionListener`
`addMouseMotionListener()`
`removeMouseMotionListener()`
	`Component` şi derivate

`WindowEvent`
`WindowListener`
`addWindowListener()`
`removeWindowListener()`
	`Window` şi derivatele, incluzând `JDialog`, `JFileDialog` şi `JFrame`

`ItemEvent`
`ItemListener`
`addItemListener()`
`removeItemListener()`
	`JCheckBox`, `JCheckBoxMenuItem`, `JComboBox`, `JList` şi toate implementările interfeţei `ItemSelectable`

`TextEvent`
`TextListener`
`addTextListener()`
`removeTextListener()`
	Orice derivată din `JTextComponent` incluzând şi `JTextField`

Tratarea evenimentului se face prin suprascrierea metodelor care sunt descrise în interfeţe şi care primesc ca unic parametru evenimentul de tipul `XXEvent` generat de componenta ascultată de ascultător. Prezentăm o listă a evenimentelor şi a metodelor corespunzătoare ce trebuie suprascrise:

Interfaţa ascultător/adaptor	Metodele interfeţei
`ActionListener`	`actionPerformed(ActionEvent)`
`AdjustmentListener`	`adjustmentValueChanged(AdjustmentEvent)`

`ComponentListener`
`ComponentAdapter`
	`componentHidden(ComponentEvent)`
	`componentShown(ComponentEvent)`
	`componentMoved(ComponentEvent)`
	`componentResized(ComponentEvent)`

`ContainerListener` `ContainerAdapter`	`componentAdded(ContainerEvent)` `componentRemoved(ContainerEvent)`
`FocusListener` `FocusAdapter`	`focusGained(FocusEvent)` `focusLost(FocusEvent)`

`KeyListener`
`KeyAdapter`
	`keyPressed(KeyEvent)`
	`keyReleased(KeyEvent)`
	`keyTypes(KeyEvent)`

`MouseListener`
`MouseAdapter`
	`mouseClicked(MouseEvent)`
	`mouseEntered(MouseEvent)`
	`mouseExited(MouseEvent)`
	`mousePressed(MouseEvent)`
	`mouseReleased(MouseEvent)`

`MouseMotionListener` `MouseMotionAdapter`	`mouseDragged(MouseEvent)` `mouseMoved(MouseEvent)`

`WindowListener`
`WindowAdapter`
	`windowOpened(WindowEvent)`
	`windowClosing(WindowEvent)`
	`windowClosed(WindowEvent)`
	`windowActivated(WindowEvent)`
	`windowDeactivated(WindowEvent)`
	`windowIconified(WindowEvent)`
	`windowDeiconified(WindowEvent)`

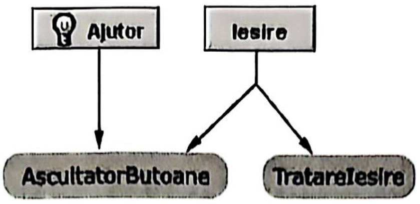
`ItemListener`	`itemStateChanged(ItemEvent)`

După cum se poate vedea şi din tabel, ascultătorul poate fi şi o extensie a unei clase adaptor, având numele `XXAdaptor`, care este o implementare abstractă a interfeţei `XXListener` şi pe care limbajul Java o pune la dispoziţie cu scopul ca programatorul să poată evita suprascrierea tuturor metodelor interfeţei. În general, adaptorii există pentru ascultătorii care au mai mult de o metodă ce trebuie implementată. Clasele adaptor le găsiţi în aceleaşi pachete unde se află şi interfeţele pe care acestea le extind. Pentru a adăuga sau şterge ascultătorii unei componente se folosesc metodele `addXXListener(XXListener ascultator)`, respectiv `removeXXListener(XXListener ascultator)`. Adăugarea de ascultători, respectiv ştergerea lor se poate realiza chiar şi dinamic, în timpul rulării aplicaţiei.

În general, componentele permit înregistrarea mai multor ascultători pentru un acelaşi eveniment (engl. *multicast mode*). Când componenta nu permite acest lucru (engl. *unicast mode*) şi forţăm ataşarea mai multor ascultători, se va arunca o excepţie `TooManyListenersException`. Nu se garantează execuţia ascultătorilor în ordinea în care au fost adăugaţi. De asemenea, un acelaşi ascultător poate fi asociat oricâtor componente care sunt surse de evenimente pe care el le poate capta. Uneori clasele care suportă evenimente de tipul `XXEvent` pun la dispoziţia utilizatorului metode de genul `fireXX()`, prin care programatorii pot genera singuri evenimente de tipul `XXEvent`, ce sunt trimise mai apoi spre tratare ascultătorilor asociaţi.

Este foarte important ca metodele ascultătorilor să se execute rapid, deoarece mânuirea evenimentelor şi desenarea componentelor sunt executate în acelaşi fir de execuţie care poartă numele de *event-despatching thread*.

Prezentăm ca prim exemplu două butoane şi tratarea operaţiei de apăsare a lor:



Tratarea a două evenimente

```
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;

@SuppressWarnings("serial")
public class TestButoane extends JFrame{

  /** Constructorul clasei */
  public TestButoane() {
    super("Test butoane");

    // stabilim gestionarul de pozitionare
    setLayout(new FlowLayout());

    // cream doua butoane
    JButton iesire = new JButton("Iesire");
    JButton ajutor = new JButton("Ajutor");

    // adaugam ascultatorul comun celor doua butoane
    iesire.addActionListener(new AscultatorButoane());
    ajutor.addActionListener(new AscultatorButoane());
    // atasarea unui nou ascultator butonului Iesire
    iesire.addActionListener(new TratareIesire());

    // adaugam butoanele in fereastra aplicatiei
    add(iesire);
    add(ajutor);

   // redimensionam fereastra, astfel incat sa se vada butoanele
    pack();
  }

  public static void main(String[] args) {
    TestButoane tb = new TestButoane();
    tb.setVisible(true);
  }
}

/**
 * Clasa ascultator care trateaza evenimentul generat
 * de apasarea unui buton. Are ca efect afisarea la consola
 * a numelui butonului.
 */
class AscultatorButoane implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    String numeButon = e.getActionCommand();
    System.out.println("Ati apasat butonul " + numeButon);
  }
}

/**
 * Clasa ascultator care are ca efect inchiderea ferestrei
 * aplicatiei din care face parte butonul apasat.
 */
class TratareIesire implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    // obtinem butonul apasat
    JButton buton = (JButton) e.getSource();

    // obtinem o instanta la fereastra aplicatiei
    JFrame fereastra =  (JFrame) buton.getTopLevelAncestor();

    // inchidem fereastra curenta
    fereastra.dispose();
  }
}
```

Putem implementa ascultătorii folosind clase anonime, aceasta fiind metoda cea mai recomandată, deoarece se compactează la un loc şi declararea ascultătorului, şi ataşarea lui componentei. Prezentăm o rescriere a ascultătorului `AscultatorButoane` din exemplul precedent:

```
iesire.addActionListener(new ActionListener()   {
  public void actionPerformed(ActionEvent e) {
    String numeButon = e.getActionCommand();
    System.out.println("Ati apasat butonul " + numeButon);
  }});
```

S-ar putea crede că în cazul ascultătorilor anonimi pierdem posibilitatea de a adăuga un acelaşi ascultător mai multor componente, dar această problemă se poate rezolva prin încapsularea funcţionalităţii ascultătorului într-o metodă aparte. Acest artificiu ne dă posibilitatea de a ascunde prin folosirea unui modificator `private` tratarea evenimentului, deoarece metodele din interfeţele ascultătorilor care trebuie suprascrise sunt, în general, publice. Astfel, putem defini metoda privată:

```
private void functionalitate(ActionEvent e) {
  String numeButon = e.getActionCommand();
  System.out.println("Ati apasat butonul " + numeButon);
}
```

Aceasta va fi apelată în metodele de tratare a evenimentului de apăsare a butonului:

```
iesire.addActionListener( new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    functionalitate(e);
  }
});

ajutor.addActionListener( new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    functionalitate(e);
  }
});
```

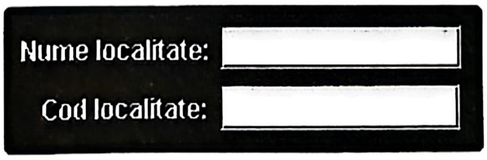
Această metodă poate fi folosită pentru a adăuga un acelaşi ascultător mai multor componente de tipuri diferite. Spre exemplu, aceeaşi operaţie de copiere (engl. *copy*) poate fi făcută şi dintr-un meniu derulant (engl. *pop-up*), ca şi dintr-o opţiune a meniului-bară, suplinind astfel folosirea clasei `AbstractAction`.

O componentă poate fi făcută ea însăşi ascultător pentru evenimentele pe care le generează sau componentele conţinute. Pentru aceasta trebuie să implementeze interfaţa corespunzătoare tipului de eveniment ascultat, să realizeze implementarea metodei care tratează evenimentul şi să-l asocieze ca ascultător. Prezentăm exemplul unei ferestre care îşi gestionează singură operaţia de închidere:

```
public class FereastraAplicatie extends JFrame implements
```

```
   ActionListener, WindowListener {
  public FereastraAplicatie() throws Exception {
    super("Fereastra");
    JButton iesire = new JButton();
    getContentPane().add(iesire);
    iesire.addActionListener(this);
    this.addWindowListener(this);
  }

  public void actionPerformed(ActionEvent e) {
    System.out.println("Ati iesit folosind butonul" +
           e.getActionCommand());
    dispose(); // inchide fereastra curenta
  }

  public void windowClosing(ActionEvent ee) {
    System.out.println("Ati iesit in mod standard");
    dispose(); // inchide fereastra curenta
  }

// obligatoriu si celelalte metode ale interfetei WindowListener
// ...
}
```

Exemplul următor prezintă o metodă prin care putem evita scrierea simultană în două zone de text, folosind evenimentul `CaretEvent`, generat în momentul schimbării poziţiei cursorului de componentele text (derivate din `JTextComponent`).



Folosirea evenimentului `CaretEvent`

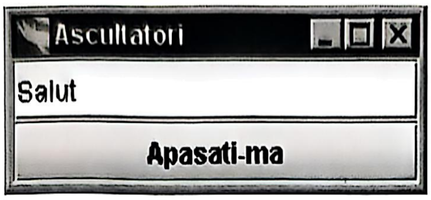
```
JTextField tloc = new JTextField(); // prima componenta text
JTextField tcod = new JTextField(); // a doua componenta text
JLabel lloc = new JLabel();
JLabel lcod = new JLabel();
//...asculta prima zona de text
tloc.addCaretListener(new javax.swing.event.CaretListener() {
  public void caretUpdate(CaretEvent e) {
    tloc_caretUpdate(e);
  }
});
//...asculta a doua zona de text
tcod.addCaretListener(new javax.swing.event.CaretListener() {
  public void caretUpdate(CaretEvent e) {
      tcod_caretUpdate(e);
    }
  });
  // tratarea miscarii cursorului prin introducerea de continut
  void tloc_caretUpdate(CaretEvent e) {
    int pozitie = e.getDot();
    if (pozitie > 0) {
      tcod.setText("");
    }
  }
  // tratarea miscarii cursorului prin introducerea de continut
  void tcod_caretUpdate(CaretEvent e) {
    int pozitie = e.getDot();
    if (pozitie > 0) {
      tloc.setText("");
    }
  }
```

Unei componente îi corespunde un număr limitat de tipuri de evenimente pe care le poate genera prin interacţiunea cu utilizatorii. Pentru a le determina trebuie să vă uitaţi la ascultătorii pe care componenta îi poate înregistra (puteţi folosi documentaţia clasei). O componentă generează numai evenimentele pentru care are ascultători înregistraţi. Spre exemplu, dacă unui câmp text (engl. *text field*) îi adăugaţi un singur ascultător care să trateze mişcările cursorului (engl. *caret*), atunci va arunca astfel de evenimente, dar nu va mai genera evenimente în momentul obţinerii focusului (deţinerea fluxului cu tastatura), apăsării butoanelor mouse-ului sau al oricărei alte acţiuni, deoarece nu are ascultători corespunzători asociaţi.

Odată ascultătorii adăugaţi unei componente, putem obţine referinţe despre ei folosind metoda pusă la dispoziţie de clasa `JComponent`, al cărei prototip este `EventListener[] getListeners(Class tipAscultator)`. Astfel, putem obţine într-un tablou toţi ascultătorii ataşaţi componentei până în momentul curent. Parametrul `tipAscultator` reprezintă tipul `ascultătorilor` ce se doresc returnaţi, o instanţă a clasei `Class` (a se vedea *Java Reflection API*), care poate fi expresia `Ascultator.class`, dacă se vor a fi returnaţi ascultătorii de tip `Ascultator`. În exemplul următor, vom adăuga un ascultător componentei, numai în cazul în care aceasta nu are deja un ascultător de acelaşi tip deja ataşat.

```
  JTextField tf = JTextField();
  MouseListener un = new MouseListener() {
    // suprascriere
  };

  MouseListener[] ml =
      (MouseListener[]) c.getListeners(MouseListener.class);
    if (ml.length == 0)
       tf.addMouseListener(un);
```

Acelaşi eveniment poate fi generat de componente diferite. Spre exemplu, `ActionEvent` poate fi generat de componenta `JButton`, dar şi de componenta `JTextField`. În cazul butonului, evenimentul este aruncat prin apăsarea lui cu mouse-ul sau prin apăsarea tastei *Space* când deţine focusul în cazul *look-and-feel*-ului obişnuit. Un *plug-in look-and-feel* poate implementa un buton care generează un astfel de eveniment atunci când utilizatorul pronunţă o anumită frază. Pentru un câmp text, ceea ce determină generarea evenimentului este apăsarea tastei *Enter*. Dacă vom asocia un acelaşi ascultător ambelor componente, ne vom folosi de informaţiile pe care evenimentul le poartă (în acest caz, sursa care l-a generat), pentru a obţine o referinţă spre sursa evenimentului. Vom folosi metoda `public Object getSource()` moştenită din clasa `EventObject` căreia îi aplicăm operatorul cast spre componenta dorită.



Fereastra aplicaţiei `Ascultatori`

```
import java.awt.GridLayout;
import java.awt.event.*;
import javax.swing.*;

public class Ascultatori extends JFrame {

  public Ascultatori() {
    super("Ascultatori");
    creeazaContinut();
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        e.getWindow().dispose();
      }
    });
    pack();
  }

  protected void creeazaContinut() {
    JPanel continut = new JPanel();
    // in acest panou vom adauga continutul ferestrei
    continut.setLayout(new GridLayout(2, 1));
    // am modificat gestionarul panoului continut
    this.getContentPane().add(continut);
    // am adaugat ferestrei panoul continut

    JTextField text = new JTextField();
    JButton buton = new JButton("Apasati-ma");
    continut.add(text);
    continut.add(buton); // am adaugat butonul
    Ascultator asc = new Ascultator(); // instantiem ascultator
    text.addActionListener(asc); // adaugam ascultator
    buton.addActionListener(asc);
  }

  public static void main(String[] args) {
    Ascultatori fereastra = new Ascultatori();
    fereastra.setVisible(true);
  }
}

class Ascultator implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    Object comp = e.getSource();
    // am obtinut componenta care a generat evenimentul
    if (comp.getClass().getName() == "javax.swing.JButton")
      System.out.println("Ati apasat butonul");
    else
      System.out.println("Ati terminat de scris");
      System.out.println("Informatii purtate de eveniment" + e);
      System.exit(0);
  }
}
```

Evenimentele pot fi împărţite în două grupe: evenimente de nivel scăzut (engl. *low level*) şi evenimente semantice. Evenimentele de nivel scăzut sunt cele determinate de dispozitivele de intrare/ieşire, cum ar fi mouse-ul sau tastatura, schimbarea dimensiunii şi poziţionării componentelor, adăugarea sau ştergerea de componente din containere, schimbarea focusului.

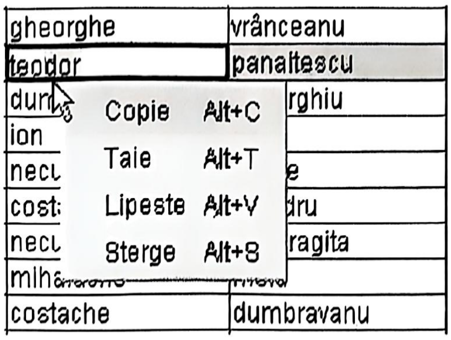
Evenimentele semantice sunt, spre exemplu, cele generate de introducerea unui caracter într-un câmp text, selectarea unui buton selectabil (engl. *checkbox*), alegerea unei opţiuni dintr-o listă, deci, în general, interacţiuni directe ale utilizatorilor cu componentele atomice. Toate evenimentele folosite până acum în exemple sunt de acest tip.

Este de preferat, atunci când se poate, ca utilizatorii să înregistreze componentelor ascultători pentru evenimentele semantice cu scopul robusteţii şi portabilităţii aplicaţiei. Spre exemplu, este de preferat să se asculte când un buton este apăsat decât să se capteze dacă butonul mouse-ului a fost apăsat în timpul în care cursorul se afla poziţionat deasupra suprafeţei butonului.

Prezentăm, în continuare, o serie de evenimente de nivel scăzut pe care le pot genera toate componentele din cauza descendenţei lor din clasa `Component` din AWT:

Nume eveniment	Nume ascultător	Nume adaptor	Explicaţie eveniment
`ComponentEvent`	`ComponentListener`	`ComponentAdapter`	Eveniment generat în momentul în care componenta îşi schimbă mărimea, poziţia sau vizibilitatea.
`FocusEvent`	`FocusListener`	`FocusAdapter`	Eveniment generat în momentul în care componenta capătă sau pierde posibilitatea de a primi comenzi de la tastatură.
`KeyEvent`	`KeyListener`	`KeyAdapter`	Eveniment generat de apăsarea unei taste doar de componentele care deţin focusul la acel moment.
`MouseEvent`	`MouseListener`	`MouseAdapter`	Eveniment generat de apăsarea butoanelor mouse-ului sau de intrarea sau ieşirea cursorului din suprafaţa componentei.
`MouseMotionEvent`	`MouseMotionListener`	`MouseMotionAdapter`	Eveniment generat de mişcarea mouse-ului în interiorul suprafeţei componentei.

Ne vom concentra, în mod special, asupra evenimentelor generate de dispozitivele standard de intrare. Presupunem cazul unei componente `JTable` tabela care prin apăsarea de două ori a butonului drept va afişa o fereastră de autentificare, iar prin apăsarea butonului drept va afişa un meniu *pop-up* (meniu contextual), permiţând operaţiile clasice de editare:



Apariţia unui meniu contextual

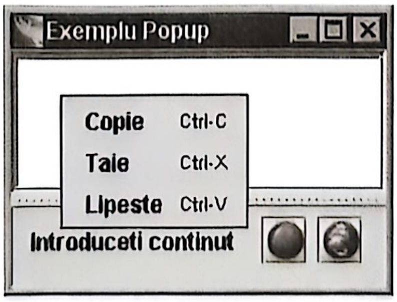
```
tabela.addMouseListener(new MouseAdapter() {
  // suprascrierea metodei care se va apela la apasarea
  // butonului mouse-ului
  public void mousePressed(MouseEvent e)   {
    int i = e.getClickCount();
    // i pastreaza numarul de apasari
    if (i == 2 && e.getButton() == 1) {
      // cazul in care am apasat de 2 ori butonul din dreapta
      Autentificare protectie = new Autentificare(null,
          "Autentificare", true);
    }
    if (e.getButton() == 3 && i == 1 && editabila = true) {
       // cazul in care am apasat butonul stang o singura data
       pop.show(e.getComponent(), e.getX(), e.getY());
       // coordonatele curente ale mouse-ului se afla cu
       // metode de tipul get()
       pop.setVisible(true);
       // afisam meniul Popup si il facem vizibil
     }
  }
});
```

În cazul unui buton *Iesire*, putem să adăugăm un ascultător care să trateze apăsarea tastei *Enter* atunci când el deţine focusul (în *look-and-feel*-ul *Metal* apăsarea tastei *Enter*, când suntem poziţionaţi pe buton, nu aruncă `ActionEvent`):

```
iesire.addKeyListener(new java.awt.event.KeyAdapter() {
  public void keyPressed(KeyEvent e) {
   int i = e.getKeyChar(); //se determină codul tastei apăsate
    if (i == 10) // daca am apasat tasta Enter
     {
         dispose();
     }
  }
});
```

Sistemul de ascultători şi evenimente asociat unei componente este gestionat prin intermediul clasei `javax.swing.event.EventManagerList` care conţine un tablou de perechi de forma `XXEvent/XXListener`. Câte o instanţă a acestei clase este asociată fiecărei componente. Modelele implicite au la rândul lor asociate astfel de instanţe. Când utilizatorul înregistrează un ascultător unei componente, aceasta adaugă în tabloul `EventManagerList` instanţa `Class` corespunzătoare evenimentului ce se vrea captat (a se vedea *Java Reflection API*). Următorul element introdus în tablou este chiar ascultătorul evenimentului, tabloul arătând astfel ca o succesiune de forma `XXEvent`, `XXListener`, `YYEvent`, `YYListener`... Atunci când un eveniment este generat, acest tablou este parcurs în mod secvenţial până la găsirea unei potriviri şi evenimentul este trimis ascultătorului corespunzător.

Putem crea ascultători complecşi pentru componente cum ar fi cel care urmează, fiind adăugat unei componente text printr-un apel `addMouseListener(new PopupSimplu())`, un ascultător determină apariţia unui meniu contextual, în care se pot face operaţii cu *clipboard*-ul sistemului de operare.



Meniu contextual pentru operaţii cu *clipboard*-ul

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import javax.swing.text.*;

class PopupSimplu extends MouseAdapter {
  JPopupMenu pup=new JPopupMenu("Operatii");
  static JMenuItem copie;
  static JMenuItem taie;
  static JMenuItem lipeste;
  JTextComponent comp;

  public PopupSimplu() {
    // cream optiunile meniului contextual care sunt instante
    // JMenuItem
    copie = new JMenuItem("Copie", new ImageIcon("c.gif"));
    // adaugam ascultator optiunii
    copie.addActionListener(new ActionListener() {
       public void actionPerformed(ActionEvent e) {
          comp.copy(); // copiem continut selectat
    }});
    pup.add(copie); // am adaugat optiunea meniului
    // adaugam acceleratori care vor aparea in dreapta
    // optiunii
    copie.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_C,
       ActionEvent.CTRL_MASK));

    taie = new JMenuItem("Taie", new ImageIcon("cu.gif"));
    taie.addActionListener(new ActionListener() {
       public void actionPerformed(ActionEvent e) {
          comp.cut(); // taiem continut selectat
    }});
    pup.add(taie);
    taie.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_X,
       ActionEvent.CTRL_MASK));

   lipeste = new JMenuItem("Lipeste", new ImageIcon("p.gif"));
    lipeste.addActionListener(new ActionListener() {
       public void actionPerformed(ActionEvent e) {
          comp.paste(); // lipim zonei text continutul
                        // clipboard-ului
    }});
    pup.add(lipeste);
    lipeste.setAccelerator(KeyStroke.getKeyStroke(
       KeyEvent.VK_V, ActionEvent.CTRL_MASK));
  }
  public void mousePressed(MouseEvent e) {
    int i = e.getClickCount();
                  // obtinem o referinta spre componenta text
    comp = (JTextComponent) e.getComponent();
    // daca am apasat o data butonul drept
    if (i == 1 && e.getButton() == 3) {
        // determinam pozitia absoluta a cursorului mouse-ului
        Point punct = e.getComponent().getLocationOnScreen();
        // vizualizam meniul Popup
        pup.show(comp, e.getX(), e.getY());
        pup.setVisible(true);
    }
  }
}
```

13.3.2. Fire de execuţie în Swing

Toate evenimentele sunt procesate de ascultători care le primesc în interiorul unui singur fir de execuţie, ce poartă numele de expeditor de evenimente (engl. *event-despatching thread*). Numele acestui fir de execuţie provine de la faptul că el are grijă ca evenimentele să ajungă la ascultători pentru a fi tratate. De asemenea, toate desenările de componente, precum şi schimbările dimensiunilor lor se vor petrece în acest fir de execuţie. *Event-despatching thread*-ul joacă un rol extrem de important în Swing şi AWT şi este cel care asigură controlul asupra schimbării stării şi modului de vizualizare a componentelor într-un mod secvenţial, nepermiţând astfel efecte neaşteptate.

Event-despatching thread are asociată o coadă (o structură de tipul FIFO – *First In First Out*) de evenimente care reprezintă, de fapt, coada de evenimente a sistemului. Orice eveniment care apare determină execuţia codului care tratează evenimentul respectiv, generând modificarea proprietăţilor componentei, a vizualizării sau redesenarea ei. Evenimentele adunate în coada de evenimente sunt procesate serial (unul după altul), pentru a evita situaţii de genul modificării stării componentei în mijlocul redesenării, ceea ce determină efecte neaşteptate. Cunoscând acest lucru trebuie să avem grijă să nu expediem evenimente în afara firului de execuţie *event-despatching thread*.

Cea mai mare problemă, în acest sens, o reprezintă modificările pe care le putem face asupra interfeţei grafice după ce aceasta a fost realizată. Pentru un container de bază cum ar fi `JFrame`, `JDialog`, a fi realizat (engl. *realized*) înseamnă că le-a fost apelată metoda `setVisible(true)` sau `pack()`. Când o componentă este realizată, toate componentele pe care le conţine se realizează. O altă metodă de a realiza o componentă este adăugarea ei într-un container deja realizat.

În general, pentru cele mai multe aplicaţii nu trebuie să purtăm grija firelor de execuţie. Este de ajuns să fie respectat următorul şablon care asigură realizarea corectă a interfeţei.

```
public static void main(String[] args) {
   Fereastra fereastra = new Fereastra();
   // ...
   // aici se adauga componente ferestrei daca nu am facut-o
   // deja in constructor
   fereastra.pack(); // aducem componentele la dimensiunea lor
     // preferata
   fereastra.setVisible(true);
   // nu se mai fac operatii asupra interfetei
}
```

Se ştie că metoda `main()` este executată într-un fir de execuţie pornit la execuţia aplicaţiei care poartă numele *main thread*. Prin apelarea unor metode de genul `setVisible()` sau `pack()`, controlul asupra interfeţei trece din *main thread* spre *event-despatching thread*, unde este executat în siguranţă.

Pentru a ne ajuta să executăm codul nostru doar din interiorul firului *event--despatching* (cu scopul de a ne proteja de evenimente neplăcute), Swing oferă o clasă numită `SwingUtilities`. Această clasă conţine două metode care ne interesează în acest context. Prima este `invokeLater()`, care permite execuţia de cod direct în firul *event-despatching*. Codul ce se vrea executat trebuie aşezat în interiorul metodei `run()` dintr-un obiect `Runnable` şi acesta trebuie introdus ca argument pentru `invokeLater()`, ca în exemplul următor:

```
Runnable deExecutat = new Runnable() {
  public void run() {
     faCeva(); // metoda care contine codul nostru
  }
};
SwingUtilities.invokeLater(deExecutat);
```

A doua metodă pe care clasa `SwingUtilities` o pune la dispoziţie este `invokeAndWait()`, care diferă faţă de precedenta prin faptul că nu se redă controlul aplicaţiei până nu se execută codul din metoda `run()`. Prezentăm în continuare un exemplu:

```
try {
  Runnable asteaptaExecutia = new Runnable() {
    public void run() {
       faCeva(); // codul ce se vrea executat
    }
  };
  SwingUtilities.invokeAndWait(asteaptaExecutia);
}
catch (InterruptedException ie) {
  System.err.println("...s-a intrerupt executia!");
}
```

```
catch (InvocationTargetException ite) {
  System.err.println("...eroare in metoda run()");
}
```

Deoarece aceste bucăţi de cod sunt plasate în coada de evenimente, ele se execută din interiorul firului *event-despatching* şi trebuie să avem grijă ca ele să se execute rapid, ca orice metodă care tratează evenimente. Dacă metoda `faCeva()` din exemplele precedente are un timp lung de execuţie, vom observa că aplicaţia îngheaţă. Pentru a rezolva astfel de probleme, putem crea propriul nostru fir de execuţie, care să realizeze diverse operaţii ce au nevoie de timp, cum ar fi conectări la reţea, încărcări de imagini etc.:

```
Thread iaTimp = new Thread() {
  public void run() {
    faCevaIndelungat(); // o actiune care ia timp
    SwingUtilities.invokeLater(new Runnable () {
      public void run() {
        // se reactualizeaza componentele
        reDeseneazaComponentele();
      }
    });
  }
};
iaTimp.start();
```

Când vorbim despre firul *event-despatching* apare noţiunea de metodă sigură (engl. *thread safe*). Câteva metode din Swing sunt construite ca fiind *thread safe* şi nu au nevoie de tratare specială. Acest lucru este, în general, anunţat în *javadoc*. Sunt câteva metode *thread safe* chiar dacă nu sunt marcate ca fiind aşa, şi anume: `repaint()`, `revalidate()` şi `invalidate()` (a se vedea subcapitolul următor).

Pentru a executa cod numai în interiorul *event-despatching thread* putem folosi următorul şablon:

```
public void codSigur() {
  if (SwingUtilities.isEventDispatchThread()) {
     //
     // ceea ce trebuie facut...
     //
  } else {
     Runnable apeleazaCodSigur =  new Runnable() {
        public void run() {
           codSigur();
        }
      };
      SwingUtilities.invokeLater(apeleazaCodSigur);
  }
}
```

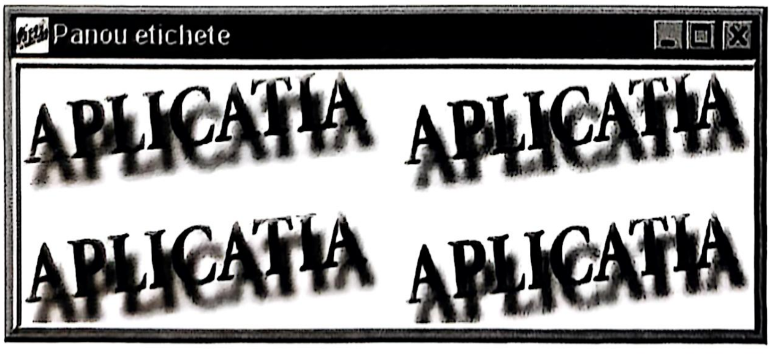
13.3.3. Desenarea componentelor grafice

Tot ceea ce este vizualizat pe ecran la un moment dat, în urma execuţiei unei aplicaţii având interfaţă grafică, rezultă în urma unui proces de desenare. Componentele sunt desenate. Procesul de desenare începe cu containerele de bază şi se continuă cu conţinutul. Componentele Swing se redesenează singure atunci când este nevoie, spre exemplu, atunci când schimbăm dimensiunile ferestrei sau când fereastra revine în prim-plan după ce a fost acoperită de o altă fereastră. Sau când apelăm pentru o etichetă o metodă de genul `setText()`, spre exemplu, eticheta se va redesena singură (conţinând noul text) şi se va şi redimensiona pentru a-l cuprinde. Utilizatorul nu trebuie să aibă în vedere desenarea doar în cazul în care doreşte să creeze propriile lui componente sau să depaneze procesul de desenare.

În Swing, pentru ca o componentă să aibă acces la desenarea sa, este necesar să suprascrie metoda `paintComponent()`, spre deosebire de AWT, unde trebuia suprascrisă `paint()`. În interiorul acestei metode avem acces la obiectul de tip `Graphics` pe care componenta îl conţine, cunoscut şi sub numele de contextul grafic al obiectului (engl. *graphic context*). Clasa `Graphics` conţine multe metode care permit desenarea figurilor geometrice sau scrierea de text cu diferite fonturi, dimensiuni şi culori.

Clasa `Graphics` foloseşte ceea ce se numeşte zona de lucru (engl. *clipping area*). În interiorul unei metode `paint()`, această zonă reprezintă regiunea din vizualizarea componentei care va fi redesenată. Mai poartă numele de regiunea murdărită (engl. *dirtied*) a componentei şi numai această zonă va fi reactualizată prin redesenarea componentei. Putem determina dimensiunile acestei regiuni folosind metoda `getBounds()`, pe care o pune la dispoziţie clasa `Graphics`. Suprafaţa de desenare se poate micşora din motive de eficienţă, spre exemplu, nefiind necesară desenarea unei anumite regiuni la un moment dat.

Prezentăm în continuare personalizarea unui panou care va conţine o etichetă cu lăţimea de 180 pixeli şi înălţimea de 60 pixeli, ce se repetă, umplându-i conţinutul:



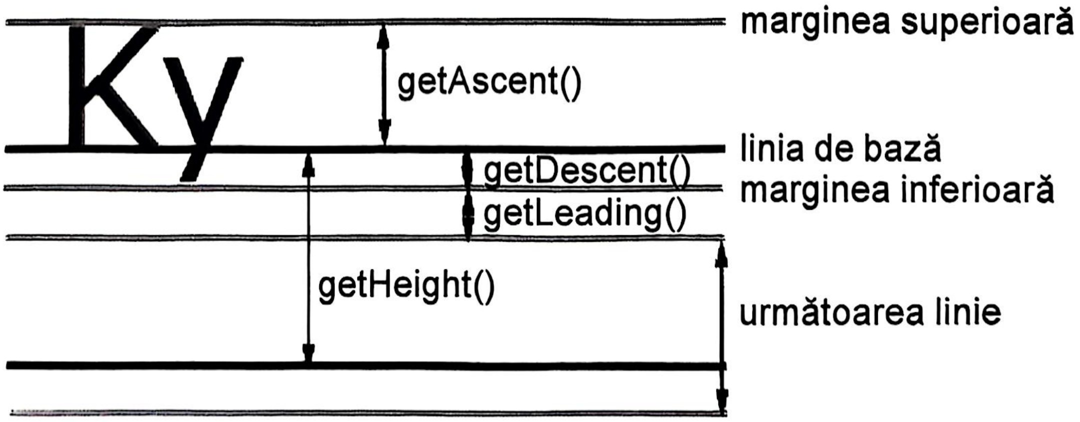
Desenarea imaginilor în fereastră

```
// personalizam clasa JPanel suprascriind
// metoda paintComponent()
JPanel ptEtichete = new JPanel() {
   public void paintComponent(Graphics g) {
      Rectangle r = g.getClipBounds(); // zona desenabila
```

```
        int i = (int) r.getHeight(); // inaltimea zonei
        int l = (int) r.getWidth(); // latimea zonei
        int nri = i/60; // de cate ori incape pe verticala
        int nrl = l/180;  // de cate ori incape pe orizontala
        for (int t = 0; t <= nrl; t++)
          for (int n = 0; n <= nri; n++)
             g.drawImage(img, t*180, n*60, null);
    }
};
```

În procesul de desenare, Swing foloseşte o tehnică care se numeşte *buffer dublu* (engl. *double-buffering*), care reprezintă desenarea într-o imagine, şi nu desenarea direct într-o componentă vizibilă pe ecran. La finalul procesului de desenare imaginea este vizualizată pe ecran (proces care se petrece rapid). Spre deosebire de Swing, în AWT dezvoltatorii erau cei responsabili pentru implementarea acestui mecanism pentru a reduce *flash*-urile. Pentru a seta/deseta acest mecanism pentru o componentă Swing se foloseşte metoda `public void setDoubleBuffered(boolean aFlag)` din clasa `JComponent`. Dacă am setat buffer-ul dublu pentru o anumită componentă, toate componentele conţinute se vor desena folosind acest mecanism.

Vom prezenta, în continuare, metodele prin care contextul grafic (în fapt, clasa `Graphics`) permite desenarea textului cu diferite fonturi, dimensiuni, culori. Mai întâi, câte ceva despre modul în care este structurat textul în cazul modului grafic pentru Java. Reperul pentru desenarea fiecărei linii de text îl reprezintă o axă orizontală care poartă numele de linia de bază (engl. *base line*). Literele care nu se prelungesc în jos (spre exemplu, litera `X`) se desenează deasupra (engl. *ascent*) acestei linii de bază. Sub linia de bază rămâne să se deseneze partea de jos a literei (engl. *descent*) pentru diacritice precum `ş` sau `ţ` ori pentru litere precum `y`. Înălţimea unui rând reprezintă distanţa dintre două linii de bază consecutive. În figura care urmează sunt prezentate aceste aspecte.



Structurarea textului

Se observă faptul că extrema de sus a unui rând este marcată printr-o linie reprezentând marginea superioară (engl. *ascender line*), iar marginea de jos este delimitată printr-o linie reprezentând marginea inferioară (engl. *descender line*). Desenarea efectivă a textului se face cu metoda `drawString(String cuvant, int x, int y)` pe care clasa `Graphics` o pune la dispoziţie. Precizăm că `x` şi `y` sunt coordonatele colţului din stânga-jos ale textului, şi, mai precis, `y` reprezintă chiar linia de bază.

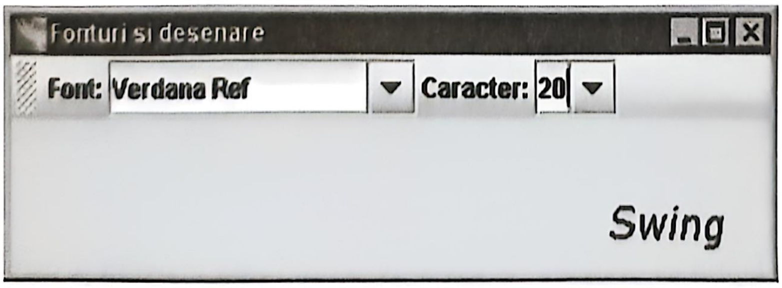
Înainte de a realiza operaţia de desenare trebuie să setăm fontul cu care se va realiza aceasta folosind metoda `setFont(Font font)` din clasa `Graphics`. Clasa `java.awt.Font` permite crearea de fonturi folosind constructori de forma `public Font(String name, int style, int size)`, unde `name` reprezintă numele fontului, de exemplu, Verdana sau Tahoma, `style` reprezintă stilul fontului care poate fi una dintre constantele `Font.BOLD`, `Font.ITALIC` sau `Font.PLAIN`, iar `size` reprezintă mărimea caracterelor.

Pentru a obţine caracteristicile fontului atribuit deja unui context grafic avem la dispoziţie clasa `FontMatrics`. Instanţa acestei clase asociată contextului grafic curent `g` se obţine printr-un apel de forma `FontMetrics fm = g.getFontMetrics()`. Prezentăm metodele pe care clasa `FontMetrics` le pune la dispoziţie pentru a obţine caracteristicile fontului, necesare în procesul de poziţionare a textului.

Prototipul metodei	Explicaţie
`int` **`getAscent`**`()`	Metoda `getAscent` returnează distanţa în pixeli dintre linia *ascender* şi linia de bază.
`int` **`getDescent`**`()`	Metoda `getDescent` returnează numărul de pixeli dintre linia de bază şi linia *descender*.
`int` **`getHeight`**`()`	Returnează numărul de pixeli dintre linia de bază a unui rând şi linia de bază a rândului următor.
`int` **`getLeading`**`()`	Returnează distanţa dintre o linie de text şi următoarea.

Culorile sunt gestionate în AWT şi Swing prin clasa `java.awt.Color` care ne permite construcţia de culori RGB prin constructori de forma: `Color(int r, int g, int b)`, unde `r`, `g`, `b` reprezintă întregi de la `0` la `255`. Clasa `Color` defineşte şi o serie de constante care reprezintă culorile simple, ca, de exemplu, `public static final Color BLACK`. Asocierea unei culori contextului grafic `g` se face printr-un apel de genul `g.setColor(Color.BLACK)`. După această asociere, metodele clasei `Graphics` vor desena cu respectiva culoare.

Prezentăm o mică aplicaţie în care se folosesc toate noţiunile discutate despre fonturi şi culori. Aplicaţia permite setarea fonturilor pe care sistemul le pune la dispoziţie şi a dimensiunilor lor din componente `ComboBox`, care apoi sunt aplicate asupra textului „Swing". Culoarea este setată semialeator folosind coordonatele obţinute prin `FontMetrics`.



Font: Verdana Ref
Caracter: 20

Modificarea fontului şi a dimensiunii acestuia

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/**
 * Fereastra pentru testarea si desenarea fonturilor
 */
@SuppressWarnings("serial")
public class TestareFonturi extends JFrame {
  /** Meniul de tip bara al aplicatiei */
  static JMenuBar meniuBara;

  /** Icon-ul ferestrei */
  final Image img;

  /** Pozitia fereastrei pe desktop */
  int pozitie = 230;

  /** Fontul utilizat. */
  Font font; // un font nou

  /** Constructorul clasei */
  public TestareFonturi() {
    super("Fonturi si desenare");
    // stabilim fontul initial
    font = new Font("Tahoma", Font.BOLD, 10);
    // pozitionam fereastra
    Dimension marimeEcran = Toolkit.getDefaultToolkit().
      getScreenSize();
    setBounds(pozitie, pozitie, marimeEcran.width - 2 * pozitie,
        marimeEcran.height / 2 - 30);
    // setam pictograma ferestrei
    Toolkit t = this.getToolkit();
    img = t.getImage("icon.jpg");
    this.setIconImage(img);
    // stabilim gestionarul de pozitionare
    this.getContentPane().setLayout(new BorderLayout());
    // cream continutul ferestrei
    creareContinut();
    // adaugam un ascultator pentru fereastra
    addWindowListener(new WindowAdapter(){
      public void windowClosing(WindowEvent e) {
        e.getWindow().dispose();
      }
    });
  }

  protected void creareContinut() {
    // cream un panou personalizat
```

```
final JPanel continut = new JPanel() {
  public void paintComponent(Graphics g) {
    String sir = "Swing";
    g.setFont(font);
    // fm va contine caracteristicile fontului
    // contextului grafic
    FontMetrics fm = g.getFontMetrics();
    int i = fm.getDescent();
    int k = fm.getAscent();
    System.out.println("i=" + i + " k=" + k);
    // cream o culoare noua (nuante inchise)
   Color culoare = new Color(i % 255, k % 255, i * k % 255);
    // setam culoarea contextului grafic
    g.setColor(culoare);
    // desenam textul
    g.drawString(sir, 300, 100);
  }
};
GraphicsEnvironment g = GraphicsEnvironment
    .getLocalGraphicsEnvironment();
// variabila fonturi va contine fonturile
// sistemului de operare
String[] fonturi = g.getAvailableFontFamilyNames();
// combobox-ul cu fonturi
final JComboBox fontBox = new JComboBox(fonturi);

fontBox.setEditable(true);
fontBox.setSelectedItem("Tahoma");
fontBox.setMaximumSize(fontBox.getPreferredSize());
String[] marimi = { "10", "15", "20", "25", "30" };

// combobox-ul cu marimea fonturilor
final JComboBox marimiBox = new JComboBox(marimi);
marimiBox.setMaximumSize(marimiBox.getPreferredSize());
marimiBox.setEditable(true);

// cream un ascultator pentru evenimente
ActionListener ascFont = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    Thread runner = new Thread() {
      public void run() {
        String str = fontBox.getSelectedItem().
          toString();
        int gr = Integer.parseInt(
          marimiBox.getSelectedItem().toString());
```

```
                font = new Font(str, Font.ITALIC, gr);
                TestareFonturi.this.repaint();
             }
          };
          runner.start();
       }
    };

    // atasam ascultatorul
    fontBox.addActionListener(ascFont);
    marimiBox.addActionListener(ascFont);

    // cream bara de unelte
    JToolBar tb = new JToolBar();
    tb.add(new JLabel(" Font: "));
    tb.add(fontBox);
    fontBox.setEditable(true);
    tb.add(new JLabel(" Caracter: "));
    tb.add(marimiBox);

    // atasam bara de unelte la fereastra
    this.getContentPane().add(tb, BorderLayout.NORTH);
    // adaugam continutul
    this.getContentPane().add(continut, BorderLayout.CENTER);
  }

  public static void main(String[] args) {
    new TestareFonturi().setVisible(true);
  }
}
```

De asemenea, pe lângă text putem să desenăm diferite figuri geometrice, de la cele mai simple la cele mai complexe. Platforma Java 2 oferă un nou pachet pentru desenare, şi anume `Java2D`, care permite următoarele: desenarea cu diverse stiluri de linii, umplerea figurilor geometrice cu texturi şi gradienţi, mutări, rotiri, scalări de text şi figuri geometrice, suprapuneri de text şi figuri, lucrul cu imagini, tipărirea.

13.3.4. Alinierea la JavaBeans

Trebuie remarcat faptul că, prin modul de construcţie, orice componentă Swing este o componentă JavaBeans, acest lucru având implicaţii importante.

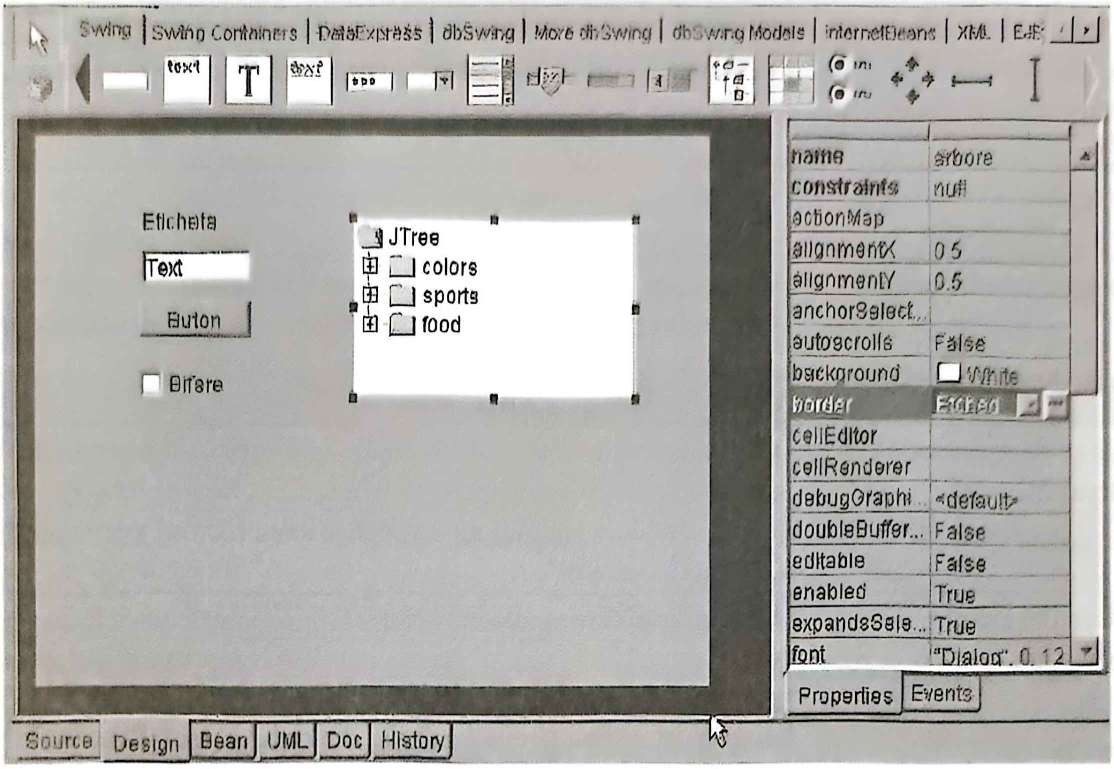
O definiţie a componentelor ar putea fi următoarea: mici bucăţi de software compilate, reutilizabile şi personalizabile, care pot fi încărcate într-un mediu de dezvoltare şi integrate împreună cu alte componente pentru a forma o nouă aplicaţie, fără a fi recompilate. De asemenea, se poate vedea componenta ca o implementare a unei funcţionalităţi de sine stătătoare. Din acest punct de vedere, funcţionalitatea fiecărei componente poate fi extrem de variată. O componentă poate să aibă o reprezentare vizuală, aşa cum este şi cazul componentelor grafice din Swing. Poate, de asemenea, să nu aibă o reprezentare vizuală, astfel de exemple fiind procesoarele de date XML, interogări SQL, baze de date etc. În Java, componentele poartă denumirea de JavaBeans, iar specificaţia JavaBeans pe care o puteţi găsi la adresa *http://www.javasoft.com/beans/docs/spec.html* stă la baza construirii componentelor în Java.

Componentele în Java sunt modelate folosind clasele. Astfel, componentele sunt clase care ţin anumite proprietăţi ce pot fi accesate numai prin metode de acces specificate. Putem gândi proprietăţile ca perechi atribut-valoare, metodele de acces permiţând citirea sau modificarea valorii unui atribut. Proprietăţile nu sunt altceva decât variabile globale ale clasei, iar dacă `numeProprietate` reprezintă numele proprietăţii, metodele de acces sunt, prin convenţie, metode având numele de forma `setNumeProprietate()`, `getNumeProprietate()`, `isNumeProprietate()`.

Proprietăţile care nu aruncă evenimente în momentul în care li se schimbă valorile poartă numele de proprietăţi simple. De asemenea, componentele pot arunca evenimente, tratabile prin *listener*-ii care le ascultă, înainte sau după ce se modifică valoarea proprietăţilor lor. Astfel, o proprietate se numeşte legată (engl. *bound property*) dacă se va arunca un eveniment `PropertyChangeEvent` după ce îşi schimbă starea. Aceste evenimente sunt ascultate de *listener*-i `PropertyChangeListener` adăugaţi componentelor folosind metoda `addPropertyChangeListener()`. O proprietate se numeşte constrânsă (engl. *constrained property*) dacă aruncă un eveniment `PropertyChangeEvent` înainte de a-şi schimba starea. Evenimentele `PropertyChangeEvent` poartă trei informaţii, şi anume: numele proprietăţii, vechea valoare şi noua valoare. Aceste evenimente sunt ascultate de *listener*-ii `VetoableChangeListener` adăugaţi componentelor folosind metoda `addVetoableChangeListener()`. Aceste evenimente şi ascultători sunt definite în pachetul `java.beans`. O mostră de tratare a evenimentelor generate de schimbarea valorii unei proprietăţi puteţi găsi în exemplul din secţiunea `JOptionPane`.

Respectând aceste convenţii în scrierea componentelor, un mediu de dezvoltare (engl. *builder*) va cunoaşte în momentul dezvoltării aplicaţiei ce proprietăţi are componenta, dacă acestea se pot doar citi sau se pot şi modifica, şi cum se pot citi şi scrie acestea. Această proprietate a componentelor de a se face cunoscute poartă numele de *introspecţie*. În fapt, nu există nici o clasă `java.beans.Bean` pe care o clasă trebuie să o extindă pentru a putea fi privită ca o componentă, ci e necesar doar ca acea clasă să respecte convenţiile din specificaţia `JavaBeans` pentru a fi considerată componentă.

Prin mediu de dezvoltare vom înţelege o unealtă (engl. *tool*) care este în fapt un mediu vizual în care componentele pot fi încărcate, personalizate şi legate între ele pentru a forma împreună aplicaţii de sine stătătoare. Pentru Java există mai multe medii de dezvoltare (engl. *IDE – Integrated Development Environment*), dintre care s-au distins câteva: *JBuilder* produs de Borland, *NetBeans* produs de Sun Microsystems, *JDeveloper* produs de Oracle, *Eclipse* oferit de IBM etc. Unul dintre beneficiile pe care îl aduc mediile integrate este posibilitatea creării rapide a interfeţelor grafice prin simpla tragere a componentelor pe suprafaţa de lucru, urmată de personalizarea lor. Timpul necesar pentru crearea unei interfeţe grafice se micşorează astfel de la câteva ore la câteva minute. Prezentăm, în continuare, o captură din mediul vizual JBuilder pentru a ilustra fereastra design în care se construiesc interfeţele grafice:



name	arbore
constraints	null
actionMap	
alignmentX	0.5
alignmentY	0.5
anchorSelect...	
autoscrolls	False
background	☐ White
border	Etched
cellEditor	
cellRenderer	
debugGraphi...	<default>
doubleBuffer...	False
editable	False
enabled	True
expandsSele...	True
font	"Dialog", 0, 12

Construirea interfeţelor grafice cu `JBuilder`

S-a constatat că este mai uşor să utilizezi o componentă creată de alţii, personalizând-o şi integrând-o cu alte componente deja existente, decât să scrii cod care să-i suplinească funcţionalitatea. Un alt beneficiu al utilizării componentelor este acela că firmele mici pot produce componente cu diverse funcţionalităţi, care vor fi cumpărate de alţii şi utilizate în aplicaţiile lor. Pe Web puteţi găsi componente grafice gata realizate la următoarele adrese Web: *http://www.componentsource.com/Marketplace/* şi *http://industry.java.sun.com/solutions*.

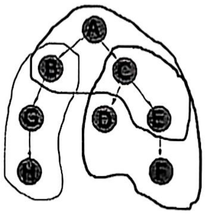
13.3.5. Gestiunea „focusului"

Prin *focus* vom înţelege proprietatea unei componente de a deţine fluxul de intrare-ieşire cu tastatura. Prin aşezarea componentelor într-un container Swing, ruta focusului este implicit de la stânga la dreapta şi de sus în jos. Gestionarea programatică a focusului se realizează prin intermediul clasei `KeyboardFocusManager`. Acest gestionar a fost construit astfel încât focusul să fie cât mai independentă de sistemul de operare peste care este instalată maşina virtuală.

Mai întâi trebuie înţelese câteva noţiuni, cu precizarea că sistemul de focus este gândit pentru AWT şi valabil şi pentru Swing:

Noţiuni	Explicaţii
Deţinător al focusului (engl. *focus owner*)	Componenta care deţine canalul de comunicare cu tastatura.
Deţinător permanent al focusului (engl. *permanent focus owner*)	Este o componentă care deţine focusul permanent. Spre exemplu, focus nepermanent au meniurile, bara de derulare dintr-o componentă `JscrollPane`, care atunci deţin focusul temporar doar pentru perioada de timp când sunt active.
Fereastra `Window` focusată (engl. *focused Window*)	Fereastra `Window` care conţine componenta ce deţine focusul.
Fereastra activă (engl. *active Window*)	Fereastra `Jframe` sau `JDialog` focusată.
Traversarea focusului	Reprezintă capacitatea de a schimba deţinătorul focusului folosind tastatura sau, programatic, fără ajutorul mouse-ului. Traversarea se poate realiza „înainte" spre următoarea componentă sau „înapoi" spre componenta anterioară.
Ciclul de traversare a focusului	O porţiune din ierarhia de componente care va fi traversată „înainte" sau „înapoi" într-un ciclu închis. Nici o altă componentă din afara ciclului nu va putea căpăta focusul prin operaţii „înainte", „înapoi".
Rădăcina ciclului focusului	Containerul care se află în vârful ierarhiei pentru un ciclu focus dat. Atunci când focusul este deţinut de o componentă din interiorul unui ciclu dat, operaţiile normale „înainte" şi „înapoi" nu pot determina ieşirea focusului din ciclul curent. Pentru a realiza acest lucru se definesc alte două operaţii, „ciclu sus" şi „ciclu jos", care permit trecerea folosind tastatura sau programatic sus sau jos în ierarhia de cicluri de traversare a focusului.

Orice fereastră obişnuită sau interioară este implicit rădăcină a unui ciclu focus. Putem privi ierarhia arborescentă de componente care alcătuieşte aplicaţia ca fiind partajată în diverse cicluri focus, ce au la rândul lor o aşezare arborescentă, aşa cum se poate vedea în figura următoare:



Organizarea focusului

Atunci când o componentă ajunge să deţină focusul sau îl pierde, este aruncat un eveniment `FocusEvent` care poate fi ascultat de un ascultător ce implementează interfaţa `FocusListener`. Prezentăm un exemplu în care un buton îşi schimbă culoarea textului în roşu atunci când obţine focusul şi în negru atunci când îl pierde:

```
final JButton anuleaza = new JButton("Anuleaz\u0103");
anuleaza.addFocusListener(new FocusListener() {
    public void focusGained(FocusEvent e) {
       // tratam obtinerea focusului
       anuleaza.setForeground(Color.red);
    }
    public void focusLost(FocusEvent e) {
      // tratam pierderea focusului
      anuleaza.setForeground(Color.black);
    }
});
```

Ordinea în care sunt parcurse componentele ciclului în procesul de traversare este determinată de diverse „politici” care sunt clase ce extind `FocusTraversalPolicy` şi se ataşează rădăcinii ciclului de transfer al focusului printr-un apel `setFocusTraversalPolicy(FocusTraversalPolicy politica)`. O enumerare a implementărilor implicite care sunt puse la dispoziţie este prezentată în continuare.

Tip „politică”	Explicaţie
`ContainerOrderFocusTraversalPolicy`	Traversarea ciclului se face în ordinea în care componentele au fost adăugate în container.
`DefaultFocusTraversalPolicy`	Traversarea ciclului se va face în aceeaşi ordine cu traversarea componentelor *peer* corespondente din sistemul de operare peste care este instalată maşina virtuală.
`SortingFocusTraversalPolicy`	Traversarea ciclului se realizează în ordinea dată în urma sortării componentelor după un anumit criteriu dat de un obiect instanţă al clasei `Comparator`.
`LayoutFocusTraversalPolicy`	Subclasă a clasei `SortingFocusTraversalPolicy` în care sortarea componentelor este făcută în funcţie de mărime, poziţie şi orientare. Componentele sunt grupate după criteriile enunţate în linii şi coloane şi traversarea se face pe linii de la stânga la dreapta şi pe coloane, de sus în jos.

Pentru interfeţele Swing care folosesc *look-and-feel*-urile standard sau un *look-and-feel* derivat din `BasicLookandfeel`, politica de traversare implicită pentru containere este `LayoutFocusTraversalPolicy`. Interfeţele AWT folosesc, implicit, `Default FocusTraversalPolicy`.

Trecerea de la o componentă la alta se realizează pentru sistemele de operare Windows şi Unix implicit prin apăsarea tastei `Tab` sau a combinaţiei `Ctrl+Tab`. Pentru a merge în sens invers va trebui să folosim implicit combinaţia de taste `Shift+Tab` sau `Ctrl+Shift+Tab`. Putem modifica tastele implicite pentru focus folosind un apel de genul `setFocusTraversalKeys(int id, Set keystrokes)`. De asemenea, putem anula efectul apăsării tastelor pentru mutarea focusului folosind metoda `setFocusTraversalKeysEnabled(boolean valoareAdevar)`.

Programatic, avem acces la controlul fluxului prin intermediul metodelor din clasa `KeyboardFocusManager` prezentate în tabelul următor. Dacă nu posedă un parametru de intrare, atunci au efect asupra componentei care deţine focusul în momentul apelării lor.

Medoda	Descriere
`void` **`focusNextComponent`**`()`	Trece focusul asupra componentei următoare.
`void` **`focusNextComponent`** `(Component c)`	Trece focusul asupra componentei `c`.
`void focus` **`PreviousComponent`**`()`	Trece focusul asupra componentei anterioare.
`void` **`focusPreviousComponent`** `(Component c)`	Trece focusul asupra componentei `c`.
`void` **`upFocusCycle`**`()`	Trece focusul asupra ciclului ascendent.
`void` **`upFocusCycle`** `(Component c)`	Trece focusul asupra componentei `c`.
`void` **`downFocusCycle`**`()`	Trece focusul asupra ciclului descendent.
`void` **`downFocusCycle`** `(Container c)`	Trece focusul asupra containerului `c`.

Dacă dorim să transferăm focusul de la o componentă la următoarea sau precedenta din ciclu, putem folosi unul dintre apelurile `transferFocus()` sau `transferFocusBackward()` din clasa `Component`. În exemplul următor, pentru o componentă facem ca apăsarea tastei `tab` să producă pierderea focusului şi nu scrierea unui `tab` ca text.

```
// ...
JTextArea text = new JTextArea();
// adaugam componentei text ascultatorul
// pentru apasarile tastelor
text.addKeyListener(new java.awt.event.KeyAdapter() {
  public void keyPressed(KeyEvent e) {
    // daca s-a apasat tasta Tab
    if (e.getKeyChar()==KeyEvent.VK_TAB) {
      // si daca este si tasta Shift apasata
      if (e.isShiftDown()) {
        // transferam focusul componentei precedente
        Continut.this.transferFocusBackward();
      } else {
       // transferam focusul componentei urmatoare din ciclu
        Continut.this.transferFocus();
      }
    }
  }
});
```

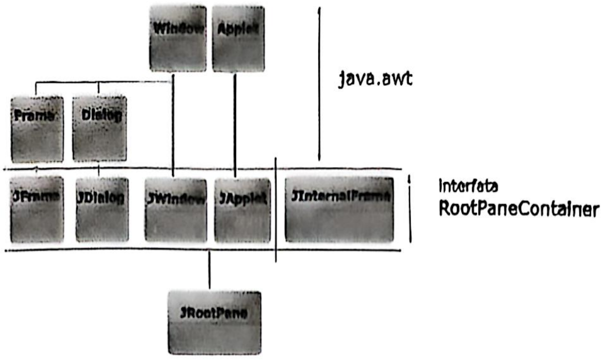
13.3.6. Clasificarea componentelor Swing

Componentele Swing pot fi împărțite în trei categorii, și anume: containere de bază, containere intermediare și componente atomice. Diferențele dintre acestea sunt prezentate în tabelul următor:

Tip componentă	Explicații	Componente
Containere de bază	Orice aplicație care deține o interfață grafică are cel puțin un astfel de container. Componentele (care nu pot fi tot containere de bază) se vor adăuga acestui container. Reprezintă locul în care toate celelalte componente se vor desena.	`JFrame, JWindow, JDialog, JApplet, JInternalFrame`
Containere intermediare	Scopul lor este acela de a grupa mai multe componente și de a simplifica poziționarea, dimensiunile și modul în care componentele conținute vor reacționa la modificarea dimensiunilor ferestrei de bază. Pentru aceasta li se atașează gestionari de poziționare (engl. *layout managers*).	`JPanel, JScrollPane, JSplitPane`
Containere atomice	Nu au, precum celelalte tipuri de componente, rolul de a ține alte componente, ci sunt suficiente pentru a prezenta informația pe care o dețin utilizatorului sau pentru a primi informații de la utilizatorii interfeței grafice. Mai pe scurt, își sunt lor suficiente pentru a interacționa cu utilizatorul.	`JLabel, JButton, JList, JComboBox, JTextField, JTextArea, JTable, JTree, JMenu, JPopup, JMenuItem, JToolBar, JOptionPane, JFileChooser, JColorChooser, JSlider, JScrollBar, JProgressBar, JDesktopBar, JInternalFrame.`

13.4. Containere de bază

Toate containerele de bază implementează interfața `RootPaneContainer` și din acest motiv au aceeași organizare internă. Diferența dintre aceste containere și celelalte componente `Swing` este faptul că ele moștenesc direct clasele corespondente lor din AWT, așa cum se poate observa în figura care urmează. Toate celelalte componente moștenesc clasa `JComponent`, și nu componentele corespondente din AWT. Din acest motiv, ele sunt singurele componente „grele" din Swing (engl. *heavyweight*), fiind indirect dependente de componentele corespondente (engl. *peer*) native ale sistemului de operare peste care este instalată platforma Java. Totuși, Swing oferă independența față de sistemul de operare de pe maşina gazdă (în sensul *look-and-feel*-ului componentelor care rămâne identic pentru toate sistemele şi maşinile) tocmai prin modul în care aceste containere sunt construite, conţinutul lor fiind gestionat de o componentă panou „uşoară", şi anume o instanţă `JRootPane`, aşa cum se va observa pe parcursul acestui capitol. Pentru alte amănunte despre diferenţele dintre Swing şi AWT, a se vedea subcapitolul AWT *vs* Swing.

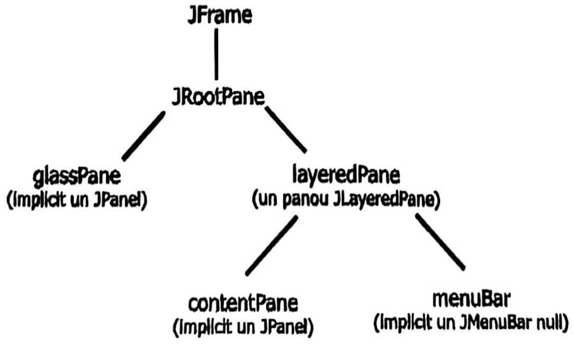

Window
Applet
java.awt
Frame
Dialog
JFrame
JDialog
JWindow
JApplet
JInternalFrame
Interfata
RootPaneContainer
JRootPane

Legătura dintre AWT şi Swing

Vom insista pe componenta `JFrame`, iar la celelalte componente `JWindow`, `JDialog`, `JApplet` vom evidenţia doar diferenţele care apar. De asemenea, vom specifica şi clasele care interacţionează cu aceste containere.

13.4.1. `JFrame`

Containerul cel mai des folosit pentru o aplicaţie bazată pe Swing este `JFrame`. Modul în care această componentă este construită este mult mai complicat decât cel al componentei corespondente din AWT, şi anume `Frame`. Orice fereastră `JFrame` conţine un câmp `protected` cu numele `rootPane`, care este o instanţă a clasei `JRootPane`, servind drept container pentru alte câteva componente panou. Nu putem adăuga componente direct unei ferestre `JFrame` (aşa cum procedam pentru o componentă `Frame`), ci unuia dintre panourile conţinute în `rootPane` (nu lui `rootPane`, care este doar un panou intermediar). Aveţi descrisă, în continuare, ierarhia implicită bazată pe relaţia *a part of* pentru o fereastră `JFrame`.


JFrame
JRootPane
glassPane
(implicit un JPanel)
layeredPane
(un panou JLayeredPane)
contentPane
(implicit un JPanel)
menuBar
(implicit un JMenuBar null)

Ierarhia implicită pentru panourile dintr-o fereastră JFrame

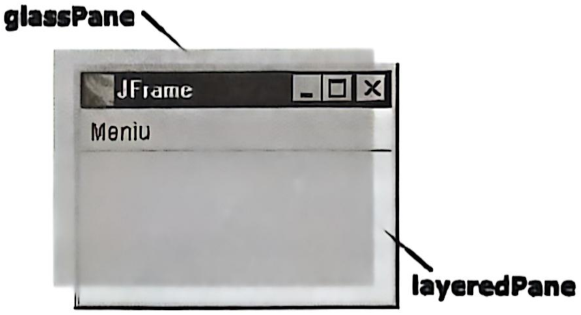
După cum se poate observa, obiectul de tip `RootPane` conţine un câmp numit `glassPane`, care este implicit un `JPanel`, şi un câmp `layeredPane`, care este o instanţă a clasei `JLayeredPane`. Acesta din urmă include, la rândul lui, un câmp `contentPane`, implicit instanţă a clasei `JPanel`, şi câmpul `MenuBar` de tip `JMenuBar`, care implicit este `null`.

Pentru a adăuga o componentă unei ferestre `JFrame` trebuie, de fapt, să ataşăm acea componentă panoului `contentPane` (putem gândi acest panou ca gestionând conţinutul ferestrei) şi acest lucru se realizează prin apeluri de genul: `fereastraMea.getContentPane().add(componentaMea)`. De asemenea, putem recurge la un apel: `JPanel continut = (JPanel) getContentPane()`, obţinând astfel panoul conţinut, căruia putem să-i adăugăm componente folosind apeluri de tipul `add()`.

Pentru a seta gestionarul de poziţionare pentru o fereastră vom folosi un apel de forma: `fereastraMea.getContentPane().setLayout(new FlowLayout())`.

Dacă vom încerca să utilizăm metode de tip `add()` sau `setLayout()` direct pentru fereastra `JFrame` vom obţine eroare. De remarcat metoda `getContentPane()`, pe care clasa `JFrame` o pune la dispoziţie pentru obţinerea unei referinţe la atributul `contentPane` asociat ferestrei.

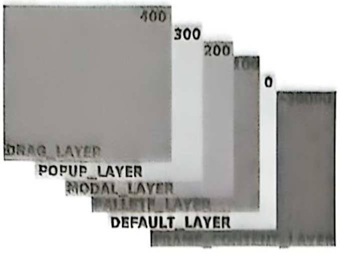
Componenta `glassPane` este implicit un panou `JPanel` neopac care stă deasupra tuturor componentelor din `JRootPane`, acţionând ca un ecran de protecţie pentru fereastră. Acest lucru permite desenarea deasupra tuturor componentelor din fereastra curentă, precum şi interceptarea sau întreruperea evenimentelor generate de mouse. Pentru a schimba `glassPane` din `JPanel` în oricare altă componentă folosim metoda `public void setGlassPane(Component glassPane)` pe care clasa `JFrame` o pune la dispoziţie. O referinţă spre componenta `glassPane` curentă asociată unei ferestre se obţine prin metoda `public Component getGlassPane()` tot din clasa `JFrame`. Implicit, `glassPane` care acoperă `layeredPane` nu este vizibilă, dar poate fi prin apelul `getGlassPane().setVisible(true)`.



Suprapunerea panoutilor `glassPane` şi `layeredPane`

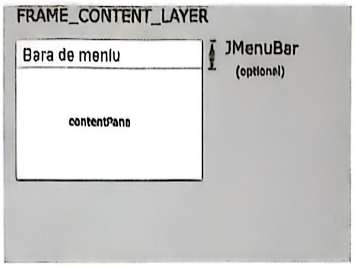
`LayeredPane` este una dintre cele mai puternice şi mai robuste componente din pachetul Swing. Reprezintă un container care oferă o a treia dimensiune pentru a poziţiona componente, şi anume adâncimea (engl. *Z-order*). Este un container având un număr nelimitat de straturi (engl. *layers*), pe care componentele pot sta. Pe un acelaşi strat putem aşeza în acelaşi timp un număr nelimitat de componente. Atunci când adăugăm o componentă unui obiect de tip `LayeredPane`, trebuie să specificăm şi adâncimea sub forma unui întreg care reprezintă numărul stratului. Cu cât numărul este mai mare, cu atât adâncimea este mai mică. Apare în acest fel fenomenul de suprapunere a componentelor conţinute, acest lucru putând fi foarte util. Spre exemplu, dorim ca un meniu *pop-up* să apară deasupra celorlalte componente conţinute de fereastră.

Clasa `JLayeredPane` oferă şase obiecte de tip `Integer` constante reprezentând cele mai utilizate straturi, având următoarele nume: `FRAME_CONTENT_LAYER`, `DEFAULT_LAYER`, `PALETTE_LAYER`, `MODAL_LAYER`, `POPUP_LAYER` şi `DRAG_LAYER`. În figura următoare se pot vedea adâncimile straturilor menţionate:


400
300
200
0
DRAG_LAYER
POPUP_LAYER
MODAL_LAYER
DEFAULT_LAYER

Adâncimile straturilor

Stratul `FRAME_CONTENT_LAYER` este locul în care panoul-rădăcină asociat fiecărei ferestre adaugă panoul conţinut (`contentPane`) şi opţional bara de meniu. Îl prezentăm în figura următoare:


FRAME_CONTENT_LAYER
Bara de meniu
JMenuBar
(opţional)
contentPane

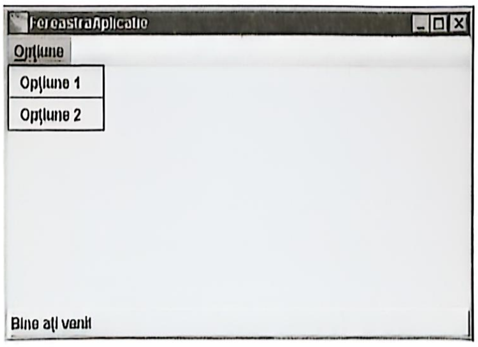
Stratul `FRAME_CONTENT_LAYER`

Bara de meniu nu există implicit, dar poate fi stabilită folosind metoda `setMenuBar` din clasa `JFrame`:

```
JMenuBar meniu = new JMenuBar();
setMenuBar(meniu);
```

Astfel, bara de meniu este poziţionată în partea de sus a stratului `FRAME_CONTENT_LAYER`, restul acestui strat fiind ocupat de panoul `ContentPane`, cum se observă în figura de mai sus. Implicit, `contentPane` este un `JPanel` opac. Poate fi setat ca fiind oricare altă componentă printr-un apel de metodă aflată în clasa `JFrame`, şi anume `setContentPane(componentaMea)`.

Prezentăm în continuare o clasă ce construieşte o fereastră, pe care o puteţi folosi ca şablon (engl. *pattern*) pentru ferestrele aplicaţiilor dumneavoastră, adaptând-o după necesităţi:


FereastraAplicatie
Opţiune
Opţiune 1
Opţiune 2

Bine aţi venit

Exemplu de fereastră principală a aplicaţiei

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;

@SuppressWarnings("serial")
public class FereastraPincipala extends JFrame {

  /** Constructorul clasei */
  public FereastraPincipala() {
    super("Fereastra Principala");
    // pozitionam fereastra si-i setam dimensiunile
    Toolkit t = this.getToolkit();
    Dimension marimeEcran = Toolkit.getDefaultToolkit().
       getScreenSize();
    int pozitie = 230;
    setBounds(pozitie, pozitie, marimeEcran.width - 2 * pozitie,
       marimeEcran.height / 2 - 30);
    // setam pictograma din coltul din stanga-sus al ferestrei
    Image img = t.getImage("icon.jpg");
    this.setIconImage(img);
    this.getContentPane().setLayout(new BorderLayout());
    // construim partile componente
    creeazaMeniu();
    creeazaContinut();
    creeazaBaraDeStare();
    // specificam modul de reactie a ferestrei la
    // inchidere (e.g. paraseste aplicatia)
    this.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        iesire();
      }
    });
  }

  /** Creeaza meniul aplicatiei */
  protected void creeazaMeniu() {
    JMenuBar meniuBara = new JMenuBar(); // cream bara de meniu
    meniuBara.setOpaque(true);
    JMenu optiuni = creeazaOptiunileMeniului(); //cream un meniu
    // optiuni
    optiuni.setMnemonic('O'); // definim shortcut (Alt+O)
    meniuBara.add(optiuni);   // adaugam meniul optiuni la
    // bara de meniu
    setJMenuBar(meniuBara);
```

```
}

/** Creeaza optiunile meniului si ataseaza ascultatorii *
protected JMenu creeazaOptiunileMeniului() {
  JMenu unelte = new JMenu("Op\u0163iune");
  JMenuItem o1 = new JMenuItem("Op\u0163iune 1");
  JMenuItem o2 = new JMenuItem("Op\u0163iune 2");
  // adaugam ascultatorul pentru optiunea1
  o1.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      // actiunea in cazul alegerii optiunii 1
    }
  });
  // adaugam ascultatorul pentru optiunea2
  o2.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      // actiunea in cazul alegerii optiunii 2
    }
  });
  // adaugam optiunile la meniu
  unelte.add(o1);
  unelte.addSeparator();
  unelte.add(o2);
  return unelte;
}

/** Creeaza continutul ferestrei */
protected void creeazaContinut() {
  JPanel continut = new JPanel(); // in acest panou vom
  // adauga continutul ferestrei
  this.getContentPane().add(continut, BorderLayout.CENTER
}

/** Creeaza bara de stare a ferestrei. */
protected void creeazaBaraDeStare() {
  JPanel ajutor = new JPanel();
  // panoul care va tine eticheta de stare
  ajutor.setBorder(new SoftBevelBorder(SoftBevelBorder.RAISED
  ajutor.setLayout(new BorderLayout());
  // punem in evidenta panoul prin bordura
  JLabel stare = new JLabel("Bine a\u0163i venit");
  // setam textul etichetei
  ajutor.add(stare, BorderLayout.CENTER);
  this.getContentPane().add(ajutor, BorderLayout.SOUTH);
  // adaugam bara de stare
  }

  /** Inchiderea aplicatiei */
  public void iesire() {
    dispose(); // eliberam resursele
    System.exit(0); // parasim aplicatia
  }
}
```

În general, o aplicaţie are o singură fereastră de bază care este apelată din interiorul clasei principale (cea care conţine metoda `main()`), aşa cum rezultă din codul următor:

```
import javax.swing.*;

public class Aplicatie {
  public static void main(String[] args) {
    try {
      UIManager.setLookAndFeel(
        "com.incors.plaf.kunststoff.KunststoffLookAndFeel");
      FereastraPincipala fereastra = new FereastraPincipala();
      fereastra.setVisible(true);
    } catch (Exception e) {
      System.err.println("Eroare: " + e);
    }
  }
}
```

13.4.2. Clase şi interfeţe legate de ferestre

Interfaţa `WindowConstants` este folosită pentru a specifica modul în care o fereastră `JFrame` sau `JDialog` reacţionează la închidere folosind metoda `setDefaultCloseOperation(int constanta)`. Sunt definite patru constante întregi a căror descriere este prezentată în tabelul următor:

Constanta	Descriere
`DISPOSE_ON_CLOSE`	Distruge fereastra şi conţinutul.
`DO_NOTHING_ON_CLOSE`	Nu reacţionează la operaţia de închidere.
`EXIT_ON_CLOSE`	Se părăseşte şi aplicaţia.
`HIDE_ON_CLOSE`	Ascunde fereastra, ea fiind uşor de revizualizat la un moment ulterior.

Interfaţa `WindowListener` trebuie implementată de toţi ascultătorii care tratează evenimentele produse de fereastra căreia i-au fost ataşaţi folosind metoda `public void addWindowListener (WindowListener l)`. Această interfaţă are şapte metode, fiecare tratând un anumit tip de eveniment, toate trebuind a fi suprascrise de către cei ce creează ascultători. Le prezentăm în continuare fără explicaţii, rolul lor înţelegându-se din denumirea pe care o poartă:

- `void windowActivated(WindowEvent e)`
- `void windowClosed(WindowEvent e)`
- `void windowClosing(WindowEvent e)`
- `void windowDeactivated(WindowEvent e)`
- `void windowDeiconified(WindowEvent e)`
- `void windowIconified(WindowEvent e)`
- `void windowOpened(WindowEvent e)`

Clasa `WindowAdapter` este o implementare abstractă a interfeţei `WindowListener`, permiţând, prin extindere, suprascrierea unui număr limitat de metode ale interfeţei `WindowListener`.

Clasa `WindowEvent` reprezintă evenimentul generat în momentul schimbării stării ferestrei (închidere, minimizare etc.). Este captat şi tratat de orice `WindowListener` sau `WindowAdapter` ataşat ferestrei tocmai în acest scop. Există şapte tipuri de evenimente, fiecare fiind tratat de metoda corespunzătoare din ascultător: `WINDOW_ACTIVATED`, `WINDOW_CLOSED`, `WINDOW_CLOSING`, `WINDOW_DEACTIVATED`, `WINDOW_DEICONIFIED`, `WINDOW_ICONIFIED`, `WINDOW_OPENED`. Prezentăm două metode ale acestei clase:

Metoda	Explicaţie
`public Window` **`getWindow()`**	Returnează fereastra-sursă a evenimentului.
`public String` **`paramString()`**	Returnează un şir de caractere care descrie tipul de eveniment, sursa şi altele; este util în procesul de depanare.

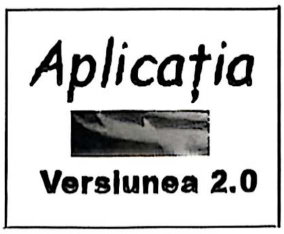
Clasa `Toolkit` permite obţinerea rezoluţiei ecranului printr-un apel de forma: `Dimension dim = getToolkit().getScreenSize()`. Implicit, fereastra se va afişa în colţul din stânga-sus al ecranului. Putem face ca fereastra curentă să apară în centrul ecranului, indiferent de rezoluţia lui, folosind un apel de forma:

```
this.setLocation(dim.width/2 - this.getWidth()/2,
  dim.height/2 - this.getHeight()/2);
```

Aceste clase şi interfeţe le vom folosi şi comenta în exemplele acestui capitol.

13.4.3. `JWindow`

Componenta `JWindow` este asemănătoare cu `JFrame`, cu excepţia faptului că nu are bară de titlu, nu i se pot modifica dimensiunile, nu se poate minimiza, maximiza şi nu poate fi închisă. `JWindow` poate fi folosită pentru a vizualiza un mesaj temporar sau siglele de la începutul utilizării aplicaţiei. Deoarece implementează interfaţa `RootPaneContainer`, necesită acelaşi comportament din partea dezvoltatorului pentru a-i accesa conţinutul. Prezentăm, în continuare, un exemplu de siglă realizată folosind `JWindow` şi vizualizată, la începutul aplicaţiei, folosind un fir de execuţie:



Exemplu de fereastră `JWindow`

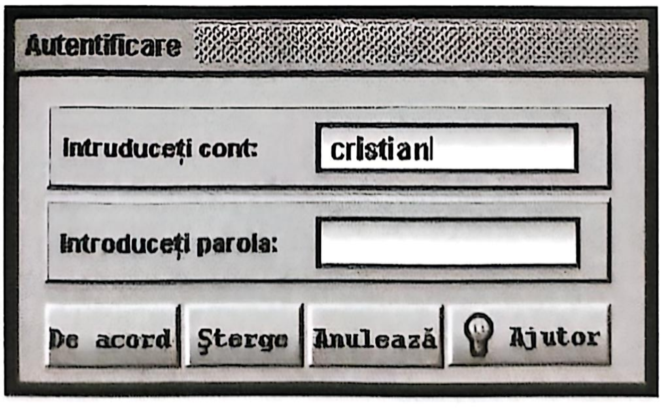
```
import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;

public class Logo {
  JWindow window;

  public Logo(JFrame f) {
    window = new JWindow(f);
    // inconjuram fereastra JWindow cu chenare pentru a nu se
    // intrepatrunde cu fundalul
    Border raisedBevel = BorderFactory.createRaisedBevelBorder();
    Border loweredBevel = BorderFactory.createLoweredBevelBorder();
    Border compus = BorderFactory.createCompoundBorder(raisedBevel,
        loweredBevel);
    JPanel panu = new JPanel(); // panoul continut
    panu.setBorder(compus);
    panu.setBackground(Color.white);
    // imaginea
    ImageIcon imaginea = new ImageIcon("logo.jpg");
    //eticheta1
    Font font1 = new Font("Comic Sans MS", Font.ITALIC, 40);
    JLabel eticheta1 = new JLabel("Aplica\u0163ia");
    eticheta1.setFont(font1);
    //eticheta2
    Font font2 = new Font("Verdana", Font.BOLD, 20);
    JLabel eticheta2 = new JLabel("Versiunea 2.0");
    eticheta2.setFont(font2);
    panu.add(eticheta1);
    panu.add(new JLabel(imaginea));
    panu.add(eticheta2);
    window.getContentPane().add(panu);
    // stabilim dimensiunile si pozitionarea ferestrei
    Dimension screenSize =
        Toolkit.getDefaultToolkit().getScreenSize();
    window.setLocation(screenSize.width/2 - 100,
        screenSize.height/2 - 130);
    window.setSize(200, 260); //dimensiuni
    window.setBackground(Color.white);
    window.setVisible(true);
    window.toFront();
  }

  // metoda care elibereaza resursele alocate ferestrei
  public void stop() {
    window.dispose();
  }
}
```

Vom activa fereastra cu sigla printr-un fir de execuţie. Sugerăm utilizarea unor modalităţi mai sofisticate pentru gestionarea timpului de expunere, ca de exemplu `java. util.Timer.`

```
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class TestLogo extends JFrame {
  boolean stoper = true;
  Logo logo;

  public TestLogo() {
    super("Test Logo");
    logo = new Logo(this);
    Thread fir = new Thread() {
      public void run() {
        try {
          while(stoper) {
            sleep(100);
          }
          sleep(6000);
        } catch (Exception e) {
          System.err.println("Eroare: " + e);
        }
        logo.stop();
      }
    };
    fir.start();
    initializareFereastra();
    stoper = false;
  }

  private void initializareFereastra() {
    JLabel label = new JLabel("  Logo Test  ");
    this.getContentPane().add(label);
    setSize(200, 100);

    this.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        e.getWindow().dispose();
      }
    });
  }

  public static void main(String[] args) {
    new TestLogo().setVisible(true);
  }
}
```

13.4.4. JDialog

Această clasă este folosită pentru a crea o fereastră de dialog ale cărei structură şi comportament sunt identice cu `JFrame`. Un dialog permite realizarea interacţiunii utilizatorilor cu aplicaţia în timpul folosirii ei, permiţând introducerea şi obţinerea de informaţii. Prin diversitatea de constructori oferiţi putem crea o fereastră `JDialog` modală sau nemodală, dependentă sau nu de un alt container. Dacă este modală, nu se permite altor ferestre să fie active în acelaşi timp cu ea. Dacă este dependentă de un părinte, dialogul se va activa doar în strânsă legătură cu el, altfel aplicaţia poate să aibă drept interfaţă grafică un simplu dialog şi nimic altceva. Puteţi consulta *javadoc*-ul pentru lista de constructori ai componentei `JDialog`. Platforma Java mai pune la dispoziţie o componentă pentru realizarea de dialoguri mai puţin complexe, şi anume `JOptionPane`, care nu face subiectul acestui capitol. Prezentăm în continuare o clasă care foloseşte un dialog modal dependent de un părinte, prin derivare din clasa `JDialog`. Prezentăm codul care generează următoarea fereastă de dialog fără partea de formatare a componentelor conţinute şi tratarea evenimentelor ce pot apărea (pe acestea le găsiţi în capitolele care descriu aceste subiecte):


Autentificare

Introduceți cont: cristian

Introduceți parola:

Fereastra de autentificare

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Autentificare extends JDialog {
  JPanel panou = new JPanel();
  JLabel parole = new JLabel();
  JLabel conte = new JLabel();
  JTextField contt = new JTextField();
  JPasswordField parolt = new JPasswordField();
  JButton deacord = new JButton();
  JButton sterge = new JButton();
  JButton anuleaza = new JButton();
  JButton ajutor = new JButton();

  public Autentificare(Frame frame, String title, boolean modal)
    super(frame, title, modal);
    try {
      initializare();
      Dimension marimeEcran = Toolkit.getDefaultToolkit().get
         ScreenSize();
      setBounds(marimeEcran.width / 2 - 150, marimeEcran.height /
         2 - 100, 407, 170);
      addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
          // eliberam resursele
          e.getWindow().dispose();
          // aici putem scrie si alte actiuni care
          // se vor realiza la inchiderea ferestrei
        }
      });
      setVisible(true);
      parolt.requestFocusInWindow();
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }

  void initializare() throws Exception {
    // desenam butoanele
    panou.setMaximumSize(new Dimension(450, 400));
    panou.setPreferredSize(new Dimension(407, 170));
    panou.setLayout(null);
    contt.setText("flavio");
    contt.setBounds(new Rectangle(139, 22, 119, 22));
```

```
        parolt.setBounds(new Rectangle(139, 63, 120, 22));
        deacord.setText("De acord");
        deacord.setBounds(new Rectangle(14, 101, 90, 28));
        deacord.setSelected(true);
        sterge.setText("\u015Eterge");
        sterge.setBounds(new Rectangle(107, 101, 90, 28));
        anuleaza.setText("Anuleaz\u0103");
        anuleaza.setBounds(new Rectangle(200, 101, 90, 28));
        ajutor.setText("Ajutor");
        ajutor.setBounds(new Rectangle(293, 101, 90, 28));
        parole.setText("  Introduce\u0163i parola:");
        parole.setBounds(new Rectangle(14, 55, 259, 37));
        conte.setText("  Introduce\u0163i contul:");
        conte.setBounds(new Rectangle(15, 14, 258, 37));
        // urmeaza formatarea butoanelor si a zonelor de text
        // si, de asemenea, tratarea evenimentelor
        panou.add(parole, null);
        panou.add(deacord, null);
        panou.add(conte, null);
        panou.add(sterge, null);
        panou.add(anuleaza, null);
        panou.add(ajutor, null);
        panou.add(parolt, null);
        panou.add(contt, null);
        this.getContentPane().add(panou, null);
    }
}
```

Pentru a obţine o fereastră de dialog de tipul celei de mai sus este nevoie de un apel de genul:

```
Autentificare protectie =
    new Autentificare(null,"Autentificare", true);
```

13.4.5. `JApplet`

Este echivalentul din Swing pentru componenta `Applet` din AWT şi permite realizarea de appleturi. Datorită faptului că pentru o componentă `JApplet` conţinutul este gestionat de un panou `JRootPane`, apare posibilitatea de a-i adăuga acesteia bară de meniu. De asemenea, se schimbă şi gestionarul de poziţionare implicit, care acum va fi `BorderLayout` (intervine în calcul panoul conţinut) în loc de `FlowLayout`. Nu vom insista asupra acestei componente datorită asemănării cu `Applet`. Pentru alte informaţii consultaţi capitolul 7, dedicat appleturilor.

13.5. Containere intermediare

Containerele intermediare le vom numi panouri (engl. *panel*, *pane*) şi sunt următoarele: `JPanel`, `JScrollPane`, `JSplitPane`, `JTabbedPane`. Panourile reprezintă modalitatea standard de a aduna mai multe componente „uşoare" la un loc şi de a le gestiona poziţionarea şi dimensiunile folosind managerii de poziţionare. Implicit, gestionarul de poziţionare pentru un panou este `FlowLayout`. De aceea, atunci când adăugăm componente într-un panou, căruia nu i-am setat alt gestionar de poziţionare, acestea se vor poziţiona pe aceeaşi linie, una după alta, cu dimensiunile lor preferate (engl. *preferred size*). Atunci când o componentă nu mai încape pe rândul curent din cauza dimensiunilor ei, se aşază pe linia următoare. Implicit, panourile nu desenează nimic altceva decât culoarea fundalului (engl. *background*) lor. Permit adăugarea de chenare şi li se poate personaliza desenarea (a se vedea capitolul 7, care descrie desenarea).

Implicit, panourile sunt opace. Mai întâi vom discuta despre chenare, gestionarii de poziţionare fiind deja discutaţi în capitolul dedicat appleturilor, pentru că sunt, de fapt, aceiaşi ca în AWT.

13.5.1. `JPanel`

Clasa `JPanel` reprezintă containerul intermediar de bază pentru orice interfaţă grafică, folosit de cele mai multe ori pentru a organiza un grup de componente ale unui alt container care le conţine. Se ştie că şi conţinutul unei ferestre este redat tot într-un panou implicit `JPanel`, numit `contentPane`, care, atenţie, este gestionat implicit de gestionarul `BorderLayout`.

Prezentăm în continuare constructorii clasei `JPanel`:

Constructor	Explicaţie
`JPanel``()`	Creează un panou.
`JPanel``(LayoutManager lm)`	Creează un panou având gestionarul de poziţionare `lm`.

Prezentăm metodele clasei `JPanel` care permit adăugarea, accesul şi ştergerea componentelor dintr-un container, adăugarea de gestionari de poziţionare, setarea chenarelor, remarcând că toate sunt moşteniri din superclasele derivate.

Metoda	Explicaţie
`void` **`add`**`(Component)` `void` **`add`**`(Component, int)`	Adaugă o componentă panoului. Dacă parametrul întreg este prezent, el indică indicele cu care componenta se adaugă în container.
`int` **`getComponentCount`**`()`	Returnează numărul de componente din panou.

`Component` **`getComponent`**`(int i)`
`Component` **`getComponentAt`**`(int x, int y)`
`Component` **`getComponentAt`**`(Point p)`
	Permit obţinerea componentei cu un anumit index, având coordonatele x, y sau aflîndu-se în punctul p.

➔

➔

`Component[] `**`getComponents`**`()`	Returnează într-un vector toate componentele containerului.

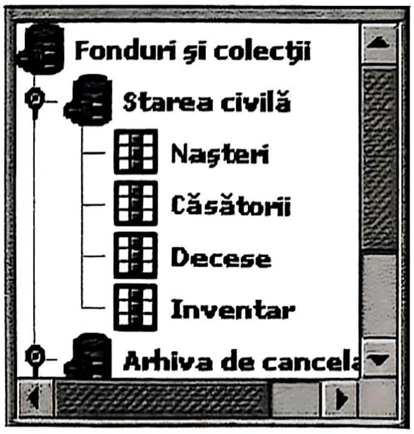
`void `**`remove`**`(Component c)`
`void `**`remove`**`(int i)`
`void `**`removeAll`**`()`
	Şterge componenta `c`, pe cea cu indicele `i` sau pe toate.

`void `**`setLayout`**`(LayoutManager lm)` `LayoutManager `**`getLayout`**`()`	Se setează, respectiv se obţine o referinţă la managerul de poziţionare.
`public void `**`setBorder`**`(Border border)`	Setează *border*-ul pentru panou.

13.5.2. `JScrollPane`

La o primă vedere această componentă pare extrem de simplă. Scopul ei este acela de a da posibilitatea derulării (engl. *scroll*) celorlalte componente în cazul în care ele nu încap într-o zonă cu dimensiuni fixe. De notat că nu permite derularea componentelor „grele" (a se vedea subcapitolele introductive). De asemenea, poziţionarea pentru această componentă este gestionată de un manager special numit `ScrollPaneLayout` şi nu se pot seta gestionarii de poziţionare obişnuiţi.

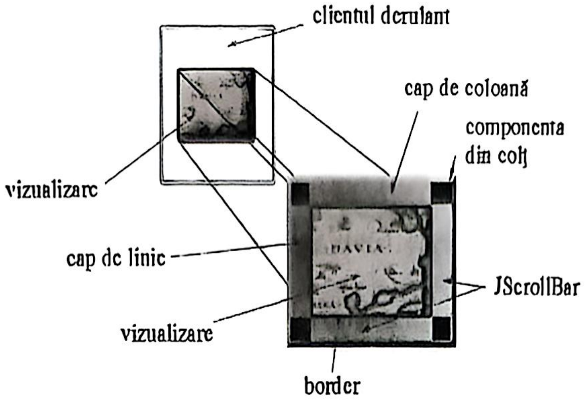
Pentru a da posibilitatea derulării unei alte componente este de ajuns să folosim un constructor al acestei clase de forma `public JScrollPane(Component view)`, aşa cum puteţi vedea în secvenţa care urmează:



Componenta `JScrollPane`

```
public JTree creeazaArbore() {
   // ...
   DefaultMutableTreeNode radacina =
      new DefaultMutableTreeNode(
         "Fonduri \u015Fi colec\u0163ii");
   radacina.add(stareaCivila = newDefaultMutableTreeNode(
        "Starea civil\u0103"));
   // ...
   return new JTree(radacina);
}

// ...
// in metoda care creeaza arborele cu bare de derulare
JTree arbore = creeazaArbore();
JScrollPane derulareArbore = new JScrollPane(arbore);
this.add(derulareArbore);
```

Modul în care este proiectată această componentă este prezentat schematic în figura următoare:



Modul de organizare a componentei `JScroolPane`

Componenta `JScrollPane` oferă o perspectivă (engl. *viewport*) asupra unei surse de date care poate fi, de exemplu, o imagine, un document text, o tabelă etc. Această sursă de date reprezintă componenta din a cărei suprafață se va vizualiza doar o anumită regiune la un moment dat, cu posibilitatea de derulare. Pentru această operație, `JScroolPane` pune la dispoziție două bare de derulare `JScrollBar`, respectiv un container `JViewport`, care reprezintă zona prin care se vizualizează componenta conținut, ce poartă denumirea de priveliște (engl. *view*) și care poate fi obținută printr-un apel `getViewport()`. Pentru a obține o referință sau pentru a seta această componentă din care doar o porțiune este vizualizată la un moment dat, putem folosi metodele `getView()`, respectiv `setView()`. Dacă nu folosiți un constructor adecvat, pentru a adăuga o componentă unui `JScrollPane` folosiți un apel de forma:

```
derulareArbore.setViewportView(arbore);
```

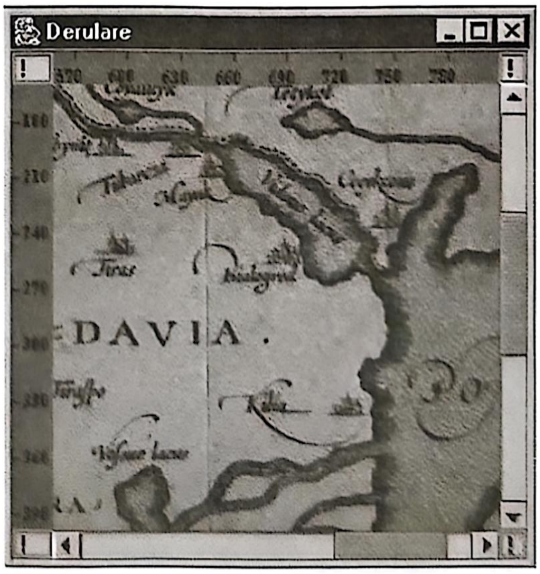
Pe lângă aceste elemente indispensabile, componenta `JScrollPane` mai pune la dispoziție un `Viewport` care are rol de cap de rând și un `Viewport` care are rol de cap de coloane. Aceste componente se vor mișca în mod sincronizat cu conținutul care se derulează. Imaginați-vă că sunt foarte utile în unele cazuri precum riglele (engl. *rulers*) în aplicațiile de desenare.

De asemenea, patru componente vor sta în colțurile rămase libere, pe care utilizatorul le poate suplini după preferință. Aceste colțuri vor fi vizibile, dacă barele de derulare și capetele care le înconjoară sunt vizibile. Folosind metoda `setCorner()`, pe care componenta `JScrollPane` o pune la dispoziție, putem să asociem o componentă colțului. Această metodă primește o componentă care va fi asignată și un șir de caractere care specifică colțul unde se va face acest lucru.

Alte metode pe care clasa `JScrollPane` le conține:

Prototipul metodei	Explicație
`public void` **`setViewport`** `(JViewport priveliste)`	Setează un nou container pentru `JScrollPane`. Va șterge vechiul `Viewport`, dacă există.
`public void` **`setViewportBorder`** `(Border bordura)`	Setează chenarul care va înconjura `ViewPort`-ul componentei `JScrollPane`.
`public void` **`setVerticalScrollBarPolicy`**`(int politica)`	Setează modalitatea în care se va face afișarea barelor de derulare. Primește drept parametri constante după cum urmează: `JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED` va determina afișarea barelor doar când este nevoie; `JScrollPane.VERTICAL_SCROLLBAR_NEVER`: nu se vor afișa niciodată barele de derulare; `JScrollPane.VERTICAL_SCROLLBAR_ALWAYS`: vor fi afișate întotdeauna.
`public void` **`setWheelScrollingEnabled`** `(boolean curotite)`	Se setează dacă se poate sau nu face *scroll* cu rotița mouse-ului.
`public void` **`setRowHeader`** `(JViewport capLinie)`	Adaugă un nou cap de linii, ștergându-l pe cel existent, dacă există. A se observa că este tot un `JViewport`.
`public void` **`setColumnHeader`** `(JViewport capColoane)`	Adaugă un nou cap de coloane, ștergându-l pe cel existent, dacă există.

Multe dintre aceste metode le veți regăsi în aplicația care urmează:



Componenta `JScrollPane` cu rigle

```
import java.awt.*;
import javax.swing.*;

public class Derulare extends JFrame {
  private JLabel eticheta;

  public Derulare() {
```

```
super("Derulare");
// imaginea care va putea fi derulata
ImageIcon icon = new ImageIcon("obiecte.gif");
eticheta = new JLabel(icon);
JScrollPane jsp = new JScrollPane(eticheta);

// cream colturile componentei de derulare
JLabel[] colturi = new JLabel[4];
for (int i = 0; i < 4; i++) {
  colturi[i] = new JLabel(" ! ");
  colturi[i].setBorder(BorderFactory.createCompoundBorder
      BorderFactory.createEmptyBorder(1, 1, 1, 1),
      BorderFactory.createLineBorder(Color.GRAY, 1)));
}

// cream linia gradata din partea stanga
JLabel capLinii = new JLabel() {
  Font f = new Font("Serif", Font.BOLD, 10);

  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    Rectangle r = g.getClipBounds();
    g.setFont(f);
    g.setColor(Color.BLUE);
    for (int i = 30 - (r.y % 30); i < r.height; i += 30)
      g.drawLine(0, r.y + i, 3, r.y + i);
      g.drawString("" + (r.y + i), 6, r.y + i + 3);
    }
  }

  public Dimension getPreferredSize() {
    return new Dimension(
      23, (int) eticheta.getPreferredSize().getHeight(
  }
};
// setam culoare de fundal si opacitatea
capLinii.setBackground(Color.GRAY);
capLinii.setOpaque(true);

// cream linia gradata de sus
JLabel capColoane = new JLabel() {
  Font f = new Font("Serif", Font.BOLD, 10);

  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    Rectangle r = g.getClipBounds();
```

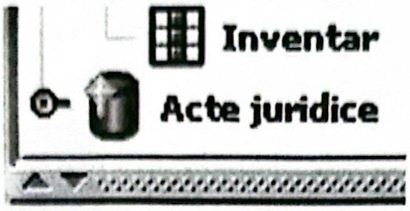
```
            g.setFont(f);
            g.setColor(Color.BLUE);
            for (int i = 30 - (r.x % 30); i < r.width; i += 30) {
               g.drawLine(r.x + i, 0, r.x + i, 3);
               g.drawString("" + (r.x + i), r.x + i - 10, 16);
            }
         }

         public Dimension getPreferredSize() {
            return new Dimension(
               (int) eticheta.getPreferredSize().getWidth(), 17);
         }
      };
      capColoane.setBackground(Color.GRAY);
      capColoane.setOpaque(true);

      // adaugam liniile gradate
      jsp.setRowHeaderView(capLinii);
      jsp.setColumnHeaderView(capColoane);

      // completam colturile
      jsp.setCorner(JScrollPane.LOWER_LEFT_CORNER, colturi[0]);
      jsp.setCorner(JScrollPane.LOWER_RIGHT_CORNER, colturi[1]);
      jsp.setCorner(JScrollPane.UPPER_LEFT_CORNER, colturi[2]);
      jsp.setCorner(JScrollPane.UPPER_RIGHT_CORNER, colturi[3]);

      // adaugam componenta
      getContentPane().add(jsp);
      setSize(300, 300);
      setVisible(true);
   }

   public static void main(String[] args) {
      new Derulare();
   }
}
```

De asemenea, `JScrollPane` este destul de maleabilă încât să ofere posibilitatea realizării operaţiei de *scroll*, fără folosirea barelor de derulare, şi glisarea componentei conţinut folosind doar mouse-ul.

13.5.3. `JSplitPane`

Componenta `JSplitPane` încapsulează două panouri alăturate separate printr-un marcaj despărţitor (engl. *divider*), permiţând vizualizarea simultană a două componente una lângă alta (sau una deasupra celeilalte). Zona despărţitoare poate fi mutată folosind mouse-ul, mărindu-se astfel spaţiul pentru unul sau altul dintre panouri, fără a modifica suprafaţa totală de vizualizare. La rândul lor, panourile unui `JSplitPane` se pot divide aşa cum se va vedea în exemplul alăturat. Acolo unde este cazul, o componentă se introduce într-un panou `JScrollPane`, care se va adăuga, mai apoi, unui panou al componentei `JSplitPane`, permiţându-i-se derularea conţinutului.

Două mici săgeţi apar într-o margine a marcajului despărţitor, aşa cum se poate vedea în captura care urmează. Acestea permit utilizatorului să extindă sau să închidă integral unul dintre panourile componentei `JSplitPane`. Modul în care apar este controlat de *plug-in*-ul pe care aplicaţia îl foloseşte. Programatorul poate seta dacă ele apar sau nu, folosind metoda `public void setOneTouchExpandable(boolean newValue)`.



Componenta `JSplitPane`

Pentru a construi un panou despărţitor putem folosi constructorul clasei `JSplitPane`:

```
JSplitPane splitPane = new JSplitPane(
    JSplitPane.HORIZONTAL_SPLIT,
    componentaStanga, componentaDreapta);
```

Primul parametru este o constantă care determină orientarea panoului, în cazul nostru pe orizontală. Dacă vrem ca panourile cele două să fie situate unul deasupra altuia putem folosi `JSplitPane.VERTICAL_SPLIT`. Orientarea poate fi schimbată şi prin intermediul metodei `public void setOrientation(int orientation)`, ai cărei parametri sunt aceleaşi constante, deja descrise.

Conţinutul panourilor unui `JSplitPane` poate fi modificat dinamic folosind metodele din tabelul următor:

Metoda	Descriere
`public void` **`setTopComponent`** `(Component comp)`	Adaugă o componentă comp în panoul de sus.
`public void` **`setLeftComponent`** `(Component comp)`	Adaugă o componentă comp în panoul din stânga.
`public void` **`setBottomComponent`** `(Component comp)`	Adaugă o componentă comp în panoul de jos.
`public void` **`setRightComponent`** `(Component comp)`	Adaugă o componentă comp în panoul din dreapta.

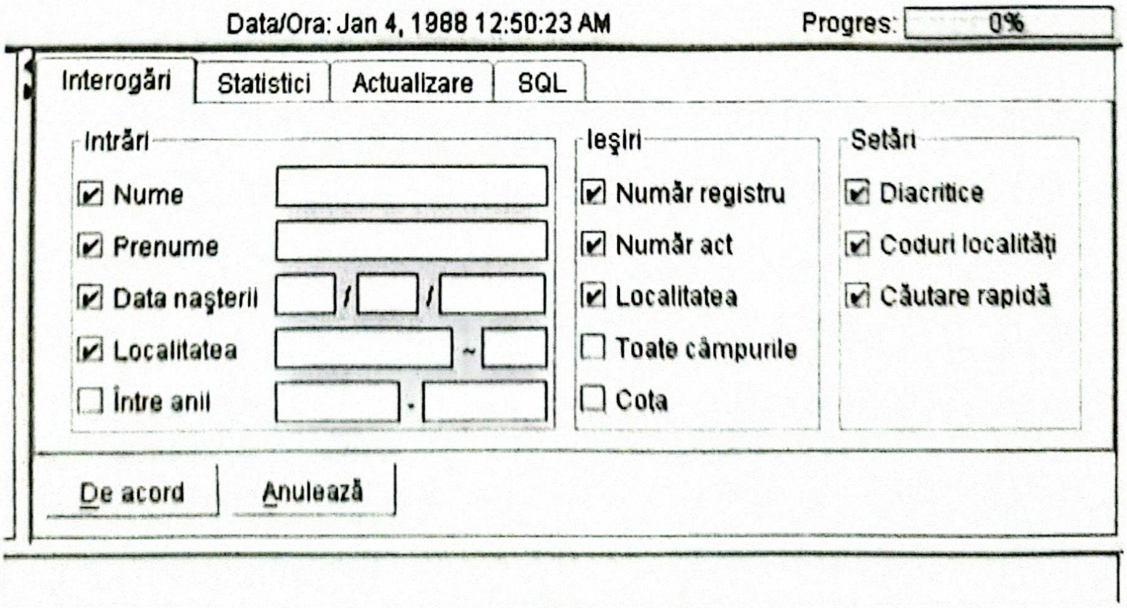
Metodele `setTopComponent()` şi `setLeftComponent()` sunt echivalente, la fel ca şi metodele `setBottomComponent()`, respectiv `setRightComponent()`, într-un anumit fel nefiind importantă orientarea.

Panourile care compun `JSplitPane` respectă dimensiunile minime ale componentelor (engl. *minimum size*) care au fost setate pentru acestea folosind `setMinimumSize()` sau prin suprascrierea metodei `getMinimumSize()` pentru componentele personalizate. De aceea, atunci când dimensiunile panourilor sunt mai mici decât dimensiunile minime ale componentei, marcajul despărţitor nu va putea fi mutat cu mouse-ul, ci eventual panourile pot fi extinse/restrânse total, folosind săgeţile ajutătoare. Pentru a controla deplasarea cu mouse-ul a marcajului despărţitor putem seta dimensiunea minimă a componentelor conţinute. De asemenea, setând dimensiunea minimă la 0 putem restrânge total dimensiunea panourilor componentei `JSplitPane`.

Poziţia marcajului despărţitor se poate schimba programatic, folosind apeluri de forma `splitPane.setDividerLocation(200)` pentru a muta la 200 de pixeli, relativ la stânga-sus sau `splitPane.setDividerLocation(0.75)` la 75%, relativ la stânga-sus. De asemenea, putem specifica pentru o componentă modul în care se va redistribui spaţiul după ce se realizează o redimensionare a ferestrei. Acest lucru se poate rezolva printr-un apel al metodei `public void setResizeWeight(double valoare)`, unde dacă valoarea este `0`, panoul din dreapta-jos va primi tot spaţiul nou, iar dacă valoarea este `1`, atunci panoul din stânga-sus va obţine tot spaţiul nou. Un exemplu de folosire a acestei componente găsiţi în secţiunea care urmează, unde o folosim în combinaţie cu `JTabbedPane`.

13.5.4. `JTabbedPane`

`JTabbedPane` reprezintă o stivă de componente (de obicei, panouri `JPanel` sau containere, în general) aşezate pe mai multe straturi suprapuse, unul dintre ele putând fi selectat la un moment dat, determinând componenta conţinută să devină vizibilă. Straturile au câte o prelungire (engl. *tab*) care are proprietăţile unei etichete (aşa cum se poate vedea în figura alăturată, unde prezentăm o astfel de componentă având patru straturi), putând conţine o pictogramă, culoare de fundal sau *tooltip*.



Data/Ora: Jan 4, 1988 12:50:23 AM

Progres: 0%

Intrări
☑ Nume
☑ Prenume
☑ Data naşterii / /
☑ Localitatea ~
☐ Între anii -

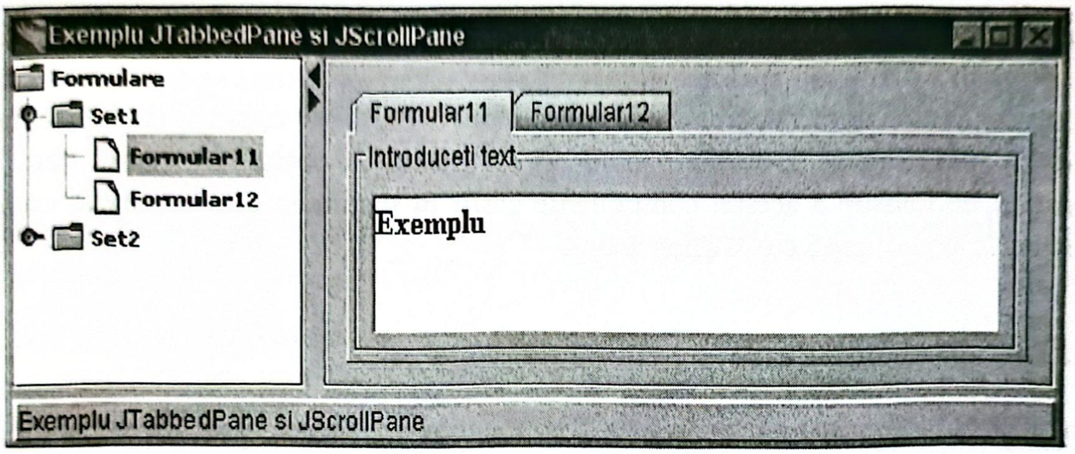
Ieşiri
☑ Număr registru
☑ Număr act
☑ Localitatea
☐ Toate câmpurile
☐ Cota

Setări
☑ Diacritice
☑ Coduri localităţi
☑ Căutare rapidă

Componenta `JTabbedPane`

Pentru a adăuga componente unui `JTabbedPane` putem folosi diverse supraîncărcări ale metodelor `add()`, `addTab()` sau `insertTab()`, pe care clasa `JTabbedPane` le pune la dispoziție. Pentru fiecare componentă adăugată unui panou `JTabbedPane` se va crea un strat unde aceasta va fi așezată; stratul va avea o prelungire cu forma unei etichete prin intermediul căreia utilizatorii vor selecta stratul pe care vor să-l vizualizeze. Fiecare strat are asociat un indice, numerotarea implicită pornind de la stânga spre dreapta, începând cu `0`. Dacă numărul de straturi este mai mare decât 0, va exista întotdeauna un indice care va fi selectat și care corespunde stratului vizibil. Implicit, este selectat stratul cu indicele `0`. Indicele curent se obține folosind metoda care are prototipul `public int getSelectedIndex()`.

Pentru a selecta programatic vizualizarea unui anumit strat vom folosi un apel `setSelectedIndex(int index)`.


Exemplu JTabbedPane si JScrollPane
Formulare
Set1
Formular11
Formular12
Set2
Formular11
Formular12
Introduceti text
Exemplu
Exemplu JTabbedPane si JScrollPane

Exemplu de utilizare a componentei `JTabbedPane`

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import javax.swing.tree.*;

public class Taburi extends JFrame {

  /** Constructorul clasei
   * initializeaza fereastra
   **/
  public Taburi() {
    super("Exemplu JTabbedPane");
    creeazaContinut();
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        e.getWindow().dispose();
        System.exit(0);
      }
    });
    setSize(500, 220);
  }

  protected void creeazaContinut() {
    final JTree arbore = creeazaArbore();

    JScrollPane panouStanga= new JScrollPane(arbore);
    panouStanga.setMinimumSize(new Dimension(150,0));

    final JScrollPane panouDreapta= new JScrollPane();
    panouDreapta.setBorder(
      BorderFactory.createCompoundBorder(
        new EtchedBorder(EtchedBorder.LOWERED),
        BorderFactory.createEmptyBorder(10,10,10,10)));
    panouDreapta.setPreferredSize(new Dimension(400,300));

    final JTabbedPane set1 = new JTabbedPane();

    JPanel formular11 = new JPanel();
    formular11.setLayout(new BorderLayout());
    final JTextArea text = new JTextArea("");

    formular11.setBorder(
      BorderFactory.createCompoundBorder(
        BorderFactory.createTitledBorder(
            BorderFactory.createEtchedBorder(),
            "Introduceti text"),
        BorderFactory.createEmptyBorder(5,5,5,5)));
    text.setBorder(BorderFactory.createLoweredBevelBorder());
    text.setWrapStyleWord(true);
    text.setFont(new Font("Serif", Font.BOLD, 16));
    text.setLineWrap(true);
    formular11.add(text);

    JPanel formular12 = new JPanel(new FlowLayout());
    JCheckBox cb1 = new JCheckBox("Camp1");
    JTextField text1 = new JTextField("Camp1");
    JCheckBox cb2 = new JCheckBox("Camp2");
    JTextField text2 = new JTextField("Camp2");

    formular12.add(cb1);
    formular12.add(text1);
    formular12.add(cb2);
```

```
formular12.add(text2);

set1.addTab("Formular11", null, formular11);
set1.addTab("Formular12", null, formular12);

final JTabbedPane set2 = new JTabbedPane();

JPanel formular21=new JPanel();
JPanel formular22=new JPanel();
set2.addTab("Formular21", null, formular21);
set2.addTab("Formular22", null, formular22);

arbore.addTreeSelectionListener(new TreeSelectionListener
  public void valueChanged(TreeSelectionEvent e) {
    DefaultMutableTreeNode nod = (DefaultMutableTreeNode
      arbore.getLastSelectedPathComponent();
    if (nod == null) return;
    String informatie = (String) nod.getUserObject();
    System.out.println(informatie);
    if(informatie == "Set1")
      panouDreapta.setViewportView(set1);
    else if(informatie == "Set2")
      panouDreapta.setViewportView(set2);
    else if(informatie == "Formular11")  {
      set1.setSelectedIndex(0);
      panouDreapta.setViewportView(set1);
    } else if(informatie=="Formular12") {
      set1.setSelectedIndex(1);
      panouDreapta.setViewportView(set1);
    } else if(informatie=="Formular21") {
      set2.setSelectedIndex(0);
      panouDreapta.setViewportView(set2);
    } else if(informatie=="Formular22") {
      set2.setSelectedIndex(1);
      panouDreapta.setViewportView(set2);
    } else
      panouDreapta.setViewportView(new JLabel());
    panouDreapta.repaint();
  }});

JPanel panouJos = new JPanel();
panouJos.setLayout(new BorderLayout());
panouJos.add(creeazaBaraStare());
panouJos.setMinimumSize(new Dimension(0,20));
panouJos.setSize(new Dimension(0,20));
```

```
  JSplitPane separatorStDr =
    new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
    panouStanga, panouDreapta);
  separatorStDr.setOneTouchExpandable(true);

  JSplitPane separatorSusJos =
    new JSplitPane(JSplitPane.VERTICAL_SPLIT,
    separatorStDr, panouJos);
  separatorSusJos.setDividerSize(5);
  separatorSusJos.setResizeWeight(1);

  final JPanel continut = new JPanel();
  // in acest panou vom adauga continutul ferestrei
  this.getContentPane().add(continut);
  continut.setLayout(new BorderLayout());
  continut.add(separatorSusJos);
}

public JTree creeazaArbore()  {
  DefaultMutableTreeNode ramura1;
  DefaultMutableTreeNode ramura2;
  DefaultMutableTreeNode radacina =
    new DefaultMutableTreeNode("Formulare");

  radacina.add(ramura1=
    new DefaultMutableTreeNode("Set1") );
  radacina.add(ramura2=
    new DefaultMutableTreeNode("Set2") );
  ramura1.add(new DefaultMutableTreeNode("Formular11"));
  ramura1.add(new DefaultMutableTreeNode("Formular12"));
  ramura2.add(new DefaultMutableTreeNode ("Formular21"));
  ramura2.add(new DefaultMutableTreeNode("Formular22"));
  // ...
  final JTree arbore = new JTree(radacina);
  DefaultTreeCellRenderer con =
    new DefaultTreeCellRenderer();
  con.setFont(new Font("Verdana",Font.BOLD,10));
  arbore.setCellRenderer(con);
  return arbore;
}

protected JPanel creeazaBaraStare() {
  // panoul care va tine eticheta de stare
  JPanel ajutor = new JPanel();
```

```
      ajutor.setBorder(
        new SoftBevelBorder(SoftBevelBorder.RAISED));
      ajutor.setLayout(new BorderLayout());
      // punem in evidenta panoul prin chenar
      JLabel stare=
        new JLabel("Exemplu JTabbedPane");
      // setam textul etichetei
      ajutor.add(stare, BorderLayout.SOUTH);
      return ajutor;
    }

    public static void main(String[] args) {
      new Taburi().setVisible(true);
    }
  }
```

13.6. Componente atomice simple

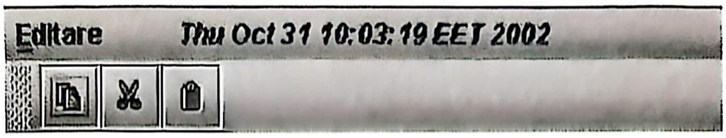
13.6.1. `JLabel`

Etichetele permit afişarea de informaţii care nu pot fi selectate sau suprascrise. De exemplu, se pot afişa text sau imagini, separat sau împreună, fără a se aştepta acţiuni din partea utilizatorilor, deoarece o etichetă nu poate deţine comunicarea cu tastatura (engl. *focus*). Pentru interacţiune se vor folosi butoane, etichetele putând, eventual, fi folosite pentru a descrie butoane sau alte componente.

Pentru a adăuga şi poziţiona conţinutul unei etichete se poate folosi unul dintre constructorii puşi la dispoziţie sau metoda `setText()` pentru a adăuga text, respectiv `setIcon()` pentru a adăuga imagini. Alinierea conţinutului unei etichete se poate seta cu una dintre metodele `setHorizontalAlignment(int aliniament)` sau `setVerticalAlignment(int aliniament)`, unde alinierea este una dintre constantele definite în interfaţa `SwingConstants`. Implicit, textul este indentat la dreapta, pe când pictogramele sunt centrate. Vertical, conţinutul este aliniat implicit centrat, indiferent dacă este text sau imagine. Pentru o etichetă se mai pot seta culoarea de fundal, culoarea textului, fontul etc., folosind metodele pe care clasa `JLabel` le pune la dispoziţie.

Un lucru interesant este faptul că o etichetă poate interpreta cod HTML afişând rezultatul obţinut. Şi alte componente Swing sunt capabile de acest lucru sau vor fi capabile în viitor. În cazul versiunii bibliotecii Swing care vine cu distribuţia 1.6 a platformei Java, se poate folosi marcaj HTML şi în cazul componentelor `JTabbedPane`, `JMenuItem`, `JToolTip`, `JRadioButton` şi `JCheckBox`.

Prezentăm, în continuare, o clasă ce suprascrie `JLabel` pentru a crea o etichetă având drept conţinut un ceas a cărui prezentare este formatată folosind HTML. Eticheta este adăugată meniului de tip bară al unei aplicaţii.



Utilizarea unei etichete în meniul de tip bară al aplicaţiei

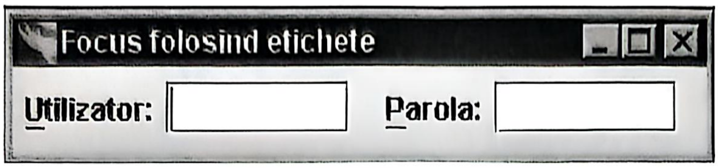
```
// Eticheta care afiseaza un ceas
import java.awt.*;
import java.awt.event.*;
import java.text.*;
import java.util.*;
import javax.swing.*;

public class CeasEticheta extends JLabel implements  Runnable {
  private Thread ceas;
  private int reactualizare = 900;

  public CeasEticheta() {
    ceas = new Thread(this);
    ceas.start();
  }
  public void reactualizeaza() {
       Date data=new Date();
       String dataNeFormatata=data.toString();
       String dataFormatata="<html> <font color=blue> <i>"+
       dataNeFormatata+"</i> </font> </html>";
       setText(dataFormatata);
  }

  // implementarea interfetei Runnable
  public void run() {
    try {
      while (true) {
        reactualizeaza();
        Thread.sleep(reactualizare);
      }
    } catch (InterruptedException e) {
    }
  }
}
```

Adăugarea etichetei meniului:

```
meniuBara.add(editare);
// adaugam meniul optiuni la bara de meniu
meniuBara.add(new JLabel("         "));
meniuBara.add(new CeasEticheta());
```

```
    setJMenuBar(meniuBara);
```

Etichetele pot fi folosite şi ca mijloc de a transfera focusul spre alte componente folosind metoda `setLabelFor()` din clasa `JLabel`. În acest sens prezentăm o metodă care setează conţinutul unei ferestre:

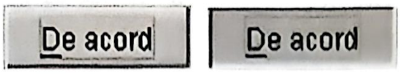
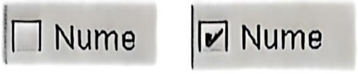
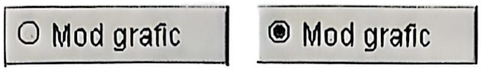


Utilizarea etichetelor pentru focus

```
protected void faContinut() throws Exception
    {
    JPanel p1 = new JPanel();
    JLabel et1 = new JLabel("Utilizator:"); // apel constructor
    et1.setDisplayedMnemonic('U');
    p1.add(et1, BorderLayout.EAST);
    JTextField te1 = new JTextField(7);
    te1.setToolTipText("Introduceti utilizator");
    et1.setLabelFor(te1);
    p1.add(te1,BorderLayout.WEST);

    JPanel p2 = new JPanel();
    JLabel et2 = new JLabel("Parola:");
    et2.setDisplayedMnemonic('P');
    p2.add(et2, BorderLayout.EAST);
    JTextField te2 = new JTextField(7);
    te2.setToolTipText("Introduceti parola");
    et2.setLabelFor(te2); // transfer focus folosind eticheta
    p2.add(te2, BorderLayout.WEST);

    JPanel continut=new JPanel(new GridLayout(0,2));
    continut.add(p1);
    continut.add(p2);

    this.getContentPane().add(continut,BorderLayout.CENTER);
    }
```

13.6.2. Butoane

Toate componentele reprezentând butoane în Swing extind clasa `AbstractButton`. De remarcat că şi clasele `JMenuItem`, `JCheckBoxMenuItem` şi `JRadioButtonMenuItem`, care desemnează opţiuni într-un meniu, extind `AbstractButton`. Prezentăm în tabelul care urmează o listă a componentelor grafice având rol de butoane, cu un exemplu de reprezentare grafică, urmat de explicaţiile corespunzătoare:

Buton	Vizualizare	Explicaţie
`JButton`		Un buton simplu a cărui apăsare determină aruncarea unui eveniment care trebuie tratat de programator. Prin apăsare, butonul îşi schimbă şi vizualizarea, iar după eliberarea butonului mouse-ului se va reveni la forma iniţială.
`JToggleButton`		Un buton care permite comutarea între două stări. Permite personalizări ale butoanelor `JCheckBox` şi `JRadioButton`, fiind şi superclasă pentru acestea.
`JCheckBox`		Un buton a cărui stare poate fi setată între bifat sau nebifat.
`JRadioButton`		Un buton radio, în general, formează împreună cu alte butoane radio un grup dintre care numai unul poate fi apăsat la un moment dat.

Stările în care se poate găsi un buton la un moment dat sunt date de următoarele proprietăţi:

Stare	Explicaţii
Selectat (engl. *selected*)	Arată dacă s-a schimbat starea butonului prin apăsarea şi apoi eliberarea butonului mouse-ului (această proprietate are valoare doar în cazul `JToggleButton` ce are două stări bifat/nebifat).
Apăsat (engl. *pressed*)	Are valoarea `true` atunci când butonul mouse-ului a fost eliberat, după ce iniţial a fost apăsat.
Derulant (engl. *rollover*)	Are valoarea `true` atunci când se tratează cazul în care cursorul mouse-ului se află deasupra suprafeţei butonului.
Armat (engl. *armed*)	Un buton se află în această stare dacă este apăsat cu mouse-ul fără a ridica degetul de pe buton; este folosită pentru a opri aruncarea evenimentului care generează tratarea, atunci când am apăsat un buton şi mai apoi îl eliberăm având cursorul mouse-ului în afara suprafeţei butonului (renunţăm să mai apăsăm butonul).
Activat (engl. *enabled*)	Are valoarea `true` când butonul poate recepta acţiunile care se fac asupra lui cu mouse-ul sau tastatura (deţine focusul) şi în acest caz celelalte proprietăţi nu pot să-şi schimbe starea când e `false`.

Dacă apăsăm cu mouse-ul un buton simplu, `JButton`, acesta se va „arma", pe când pentru un `JToggleButton`, acesta va deveni selectat.

Butoanele folosesc un model pentru a gestiona stările în care se pot afla la un moment dat. Acesta poartă numele `ButtonModel` şi reprezintă baza pentru realizarea de modele pentru butoane. Putem avea acces la acest model folosind metodele `getModel()`, respectiv `setModel()`, prima pentru a obţine modelul, iar a doua pentru a seta un nou model pentru butonul curent. Dacă nu precizăm nici un model, implicit se va asocia `DefaultButtonModel`, care reprezintă implementarea implicită ce ne este pusă la dispoziţie pentru modelul `ButtonModel`. În general, în practică, nu este nevoie să creăm propriul model pentru butoane, fiind de ajuns să folosim unul dintre constructorii puşi la dispoziţie.

13.6.2.1. *JButton*

Un buton poate prezenta atât text, cât şi imagini, toate aspectele referitoare la acest subiect fiind comune cu cele discutate în cazul etichetelor. O imagine poate fi ataşată unui buton folosind constructorul care are ca parametru o instanţă a clasei `Icon` sau, după desenare, folosind metoda `setIcon(Icon imagine)`. Putem adăuga pictograme care să apară în diferite stări ale butonului, când este selectat, apăsat, dezactivat etc.

Unui buton i se poate asocia drept accelerator o literă din textul pe care îl prezintă drept conţinut (aceasta va deveni subliniată) folosind un apel de genul `setMnemonic('B')`. În Windows, apăsarea tastei `ALT` şi a tastei reprezentând litera respectivă determină declanşarea acţiunii.

Tratarea evenimentelor generate de butoane se face în funcţie de tipul de buton. În general, se adaugă ascultători `ActionListener` care tratează evenimentul aruncat de apăsarea butonului. În cazul `JCheckBox` şi `JRadioButton` se foloseşte un ascultător `ItemListener` pentru a testa dacă butonul este sau nu selectat. Pentru exemple de implementare a ascultătorilor puteţi consulta subcapitolul referitor la evenimente şi ascultători.

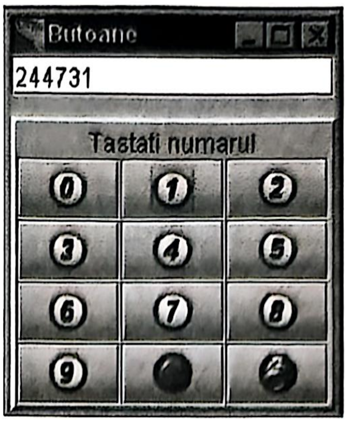
Un singur buton al interfeţei grafice poate deţine legătura cu tastatura la un moment dat, ceea ce înseamnă că apăsarea tastei spaţiu (engl. *spacebar key*) în acel moment determină declanşarea acţiunii asociate butonului. Pentru a selecta butonul implicit care deţine focusul la startarea aplicaţiei, folosim un apel `fereastraMea.getRootPane().setDefaultButton(buton)`. Un buton poate fi dezactivat (nu va mai răspunde la acţiunile utilizatorilor efectuate asupra lui) folosind metoda `setEnabled(boolean stare)`.

13.6.2.2. *JToggleButton*, *JCheckButton*, *JRadioButton*

Componentele `JCheckButton`, `JRadioButton` şi `JToggleButton` au aceeaşi funcţionalitate, permiţând simularea unui comutator (engl. *switch*) cu două stări. Singura diferenţă dintre toate aceste componente o reprezintă instanţele `UIDelegate` care le dau vizualizarea şi care sunt diferiţi. De remarcat faptul că `JCheckButton` şi `JRadioButton` moştenesc clasa `JToggleButton`, care oferă pentru acestea mecanismul de comutare între cele două stări posibile. Putem verifica dacă un buton este selectat (spre exemplu, pentru un buton `JCheckButton` acest lucru înseamnă că e bifat) folosind metoda `isSelected()` şi îl putem selecta efectiv utilizând metoda `setSelected (boolean stare)`.

Folosind clasa `ButtonGroup` putem grupa mai multe butoane împreună pentru a garanta că numai unul poate fi selectat la un moment dat. Gruparea se poate face doar pentru butoanele care moştenesc `JToggleButton`, deoarece numai acestea sunt selectabile.

Exemple de folosire a butoanelor găsiți în majoritatea aplicațiilor acestui capitol. Vom prezenta câteva metode în care sunt cuprinse multe dintre aspectele discutate în această secțiune. Acestea permit atașarea drept conținut pentru o fereastră `JFrame` a unui panou pentru introducerea numerelor.



244731

Fereastră cu butoane

```
// construim continut
protected void doContinut() throws Exception
  {
  JPanel panouSus= new JPanel();
  panouSus.setLayout(new BorderLayout());
  ecran=new JTextField();
  panouSus.add(ecran);
  panouSus.setMinimumSize(new Dimension(500,20));

  JSplitPane separatorSusJos=new JSplitPane(
   JSplitPane.VERTICAL_SPLIT, panouSus, doPanouJos());

  separatorSusJos.setResizeWeight(0);

  final JPanel continut= new JPanel();
  // in acest panou vom adauga continutul ferestrei
  this.getContentPane().add(continut);
  continut.setLayout(new BorderLayout());
  continut.add(separatorSusJos);
  }
protected JPanel doPanouJos()
  // construieste panou butoane
  {
  JPanel panou= new JPanel();
  panou.setBorder(new
    SoftBevelBorder(SoftBevelBorder.RAISED));
  panou.setLayout(new BorderLayout());
  JLabel mesaj=new JLabel("Tastati numarul",
    SwingConstants.CENTER);
  panou.add(mesaj,BorderLayout.NORTH);

  JPanel tastatura=new JPanel();
  tastatura.setLayout(new GridLayout(0,3));
  Ascultator asc=new Ascultator() // definire ascultator
  {
  public void actionPerformed(ActionEvent e)
  {
        String sir=ecran.getText();
        String nou=e.getActionCommand();
        int nr=new Integer(nou).intValue();
        if (nr<10)
           ecran.setText(sir+nou);
        // introducere numar in zona editare
        else
        if (nr==10)
           ecran.setText(""); // reseteaza text
        else
           iesire(); // parasirea ferestrei curente
  }
  };

  for(int i=0;i<12;i++)
  {
        final int aux=i;
        JButton tasta=new JButton(
         new ImageIcon("C"+aux+".gif"))
         {
            public String getActionCommand()
            {
                  return new Integer(aux).toString();
                  }
         };
        tastatura.add(tasta); // adaugam buton panoului
        tasta.addActionListener(asc);
         // atasam ascultator butonului
           }

        panou.add(tastatura);
        return panou;
        }

public void iesire() {
  System.exit(0); // parasim aplicatia
}
```

13.6.3. Chenare

Rolul chenarelor (engl. *borders*) este important în realizarea unei interfeţe grafice prietenoase. Prin intermediul lor putem crea butoane care să se distingă de context, umbre pentru zonele de text, putem separa între ele componentele. Chenarele nu sunt componente (nu moştenesc `JComponent`), astfel încât nu le putem asocia ascultători, *tooltips* etc. Clasa din care sunt derivate toate chenarele, şi anume `AbstractBorder`, este derivată direct din clasa `Object`, la fel ca toate clasele din Java. Clasele reprezentând diverse chenare se află localizate în pachetul `javax.borders` care trebuie importat pentru a le putea folosi.

Swing pune la dispoziţia utilizatorilor o serie de chenare standard care pot fi folosite în obţinerea de chenare compuse, personalizate. Lista lor o prezentăm în continuare:

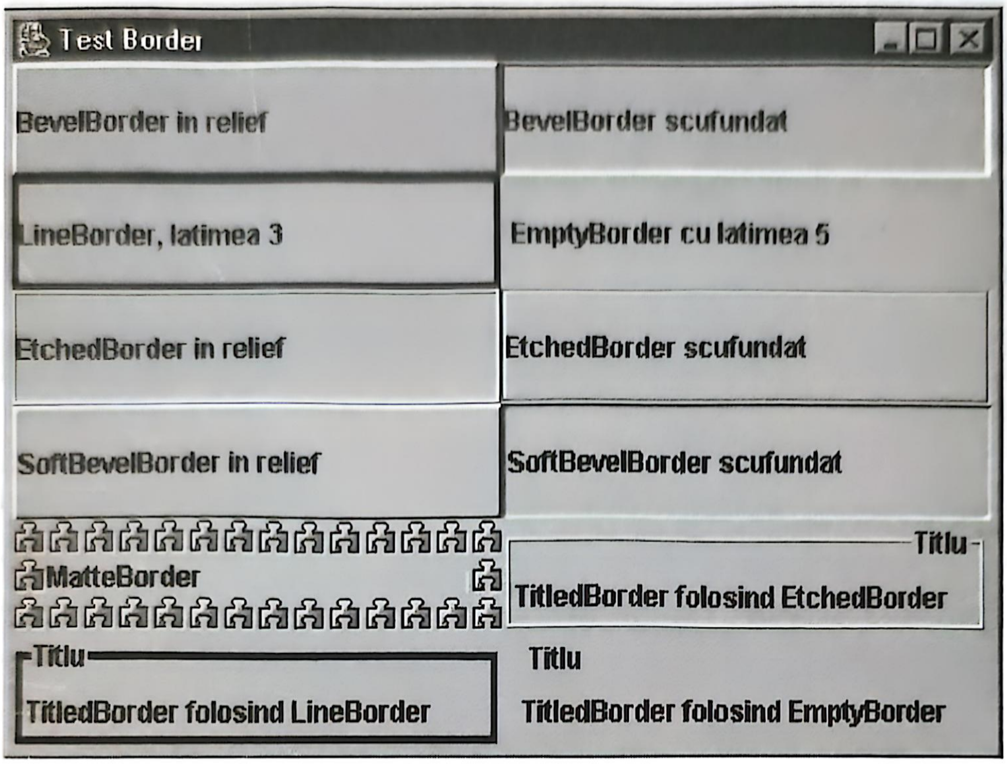
Chenar	Explicaţia
`CompoundBorder`	O combinaţie de două chenare: una interioară şi una exterioară. Reprezintă metoda prin care se pot compune chenare.
`EmptyBorder`	Un chenar transparent folosit doar pentru a defini spaţiul liber în jurul componentelor.
`EtchedBorder`	Un chenar având forma unei linii care apare în relief.
`LineBorder`	Un chenar plan cu o grosime şi culoare specificate.
`MatteBorder`	Un chenar constând dintr-o culoare sau o mică imagine.
`SoftBevelBorder`	Un chenar 3D ridicat sau coborât, cu colţurile rotunjite.
`TitledBorder`	Un chenar care afişează un titlu într-o anumită poziţie. Putem specifica fontul titlului, culoarea, indentarea şi poziţionarea folosind metodele şi constantele clasei `TitleBorder`.

Pentru a seta chenarul unei componente se utilizează metoda `setBorder()` pe care acestea o moştenesc din clasa `JComponent`. Mai există, de asemenea, o clasă care poartă denumirea de `BorderFactory`, care se află în pachetul `javax.swing`, ce conţine metode statice - pot fi folosite pentru a construi chenare de diferite tipuri. Spre exemplu, secvenţa următoare ataşează un chenar gol panoului `panou`, distanţând-ul astfel cu 10 pixeli de celelalte componente care sunt aşezate lângă el.

```
import javax.borders.*;
//...
JPanel panou = new JPanel();
panou.setBorder(BorderFactory.createEmptyBorder(
                                   10, //sus
                                   10, //stanga
                                   10, //jos
                                   10) //dreapta
                                   );
```

În general, clasele reprezentând chenare nu conţin metode pentru modificarea dimensiunilor, culorilor etc. Pentru aceasta, trebuie creată o nouă instanţă a chenarului care este asociat componentelor. De asemenea, se poate ataşa un acelaşi chenar mai multor componente.

Prezentăm în continuare o aplicaţie care ilustrează câteva tipuri de chenare pe care le vom adăuga unor butoane (acelaşi lucru se poate face şi pentru panouri, metoda `setBorder()` fiind moştenită pentru toate componentele din clasa `JComponent`):


Test Border
BevelBorder in relief
BevelBorder scufundat
LineBorder, latimea 3
EmptyBorder cu latimea 5
EtchedBorder in relief
EtchedBorder scufundat
SoftBevelBorder in relief
SoftBevelBorder scufundat
MatteBorder
Titlu
TitledBorder folosind EtchedBorder
Titlu
TitledBorder folosind LineBorder
Titlu
TitledBorder folosind EmptyBorder

Exemple de chenare

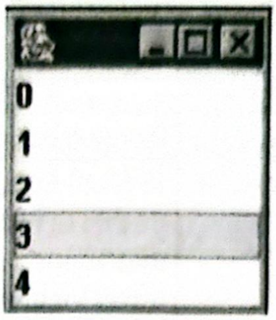
```
import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;

class TestBorder extends JFrame
{
  public TestBorder() {
    setTitle("Test Border");
    setSize(200, 400);

    JPanel cadru = (JPanel)getContentPane();
    cadru.setLayout(new GridLayout(6,2));

    JButton p = new JButton();
    p.setBorder(new BevelBorder (BevelBorder.RAISED));
    p.add(new JLabel("BevelBorder in relief"));
    cadru.add(p);
```

```
p = new JButton();
p.setBorder(new BevelBorder (BevelBorder.LOWERED));
p.add(new JLabel("BevelBorder scufundat"));
cadru.add(p);

p = new JButton();
p.setBorder(new LineBorder (Color.blue, 3));
p.add(new JLabel("LineBorder, latimea 3"));
cadru.add(p);

p = new JButton();
p.setBorder(new EmptyBorder (5,5,5,5));
p.add(new JLabel("EmptyBorder cu latimea 5"));
cadru.add(p);

p = new JButton();
p.setBorder(new EtchedBorder (EtchedBorder.RAISED));
p.add(new JLabel("EtchedBorder in relief"));
cadru.add(p);

p = new JButton();
p.setBorder(new EtchedBorder (EtchedBorder.LOWERED));
p.add(new JLabel("EtchedBorder scufundat"));
cadru.add(p);

p = new JButton();
p.setBorder(new SoftBevelBorder (SoftBevelBorder.RAISED));
p.add(new JLabel("SoftBevelBorder in relief"));
cadru.add(p);

p = new JButton();
p.setBorder(new SoftBevelBorder
  (SoftBevelBorder.LOWERED));
p.add(new JLabel("SoftBevelBorder scufundat"));
cadru.add(p);

p = new JButton();
p.setBorder(new MatteBorder (
  new ImageIcon("imprejur.gif")));
p.add(new JLabel("MatteBorder"));
cadru.add(p);

p = new JButton();
TitledBorder b=new TitledBorder(new EtchedBorder(2),
        "Titlu");
    b.setTitleJustification(TitledBorder.RIGHT);
    p.setBorder(b);
    p.add(new JLabel("TitledBorder folosind EtchedBorder"));
    cadru.add(p);

    p = new JButton();
    p.setBorder(new TitledBorder ( new LineBorder (Color.RED,
      3),"Titlu"));
    p.add(new JLabel("TitledBorder folosind LineBorder"));
    cadru.add(p);

    p = new JButton();
    p.setBorder(new TitledBorder ( new EmptyBorder (5,5,5,5
      "Titlu"));
    p.add(new JLabel("TitledBorder folosind EmptyBorder"));
    cadru.add(p);

    setVisible(true);
  }

  public static void main(String args[]) {
    new TestBorder();
  }
}
```

13.6.4. `JList`

Această componentă permite utilizatorilor să aleagă un element dintr-o listă de obiecte. Conţinutul unei liste este gestionat printr-un model care este o instanţă a clasei `ListModel`. Putem preciza că `JList` oferă acces la conţinutul listei doar prin intermediul modelului de date asociat. Constructorii acestei componente permit crearea de liste direct dintr-un tablou având drept elemente instanţe de clase derivate din clasa `Object` sau dintr-un vector `Vector` (care, la rândul lui, este o colecţie de elemente `Object`). Ceea ce va fi afişat ca text în listă, ţinând cont că elementele listei sunt instanţe `Object`, va fi şirul rezultat din apelul `toString()` al obiectelor (numai în cazul instanţelor clasei `Icon` se va face tipărirea lor ca pictograme în etichete). În cazul în care nu asociem un model de date explicit, conţinutul este gestionat tot prin intermediul unui model `ListModel` implicit, şi anume o instanţă a `DefaultListModel`, care este alocată componentei. Clasa `DefaultListModel` este o implementare implicită a modelului `ListModel` şi ţine în interior lista sub forma unui vector de obiecte `Object`. Mai există o implementare abstractă pentru `ListModel`, şi anume `AbstractListModel`, care permite programatorului să aleagă structura de date în care să păstreze lista.

Un exemplu de listă creată folosind un tablou regăsiți în capitolul în care se enumeră componentele. Prezentăm un exemplu în care construim o listă dintr-un `Vector`.



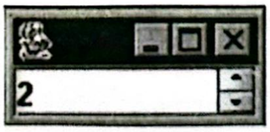
Exemplu de listă

```
Vector v=new Vector();
    for(int i=0;i<5;i++)
        {
        v.add(i,new Integer(i).toString());
        }
    JList lista = new JList(v);
```

Aceeași vizualizare o obținem prin crearea unei liste având următorul model de date:

```
ListModel model = new AbstractListModel() {
    // este de ajuns implementarea acestor doua metode pentru a
    // crea un model AbstractListModel
    public int getSize() { return 5; }
    public Object getElementAt(int index) { return new
    Integer(index).toString(); }
    };
JList lista = new JList(model);
```

Pentru a putea derula conținutul unei liste trebuie, mai întâi, să o includem într-un `JscrollPane`, printr-un apel de forma `JScrollPane derulare = new JScrollPane(lista);`



Exemplu de listă derulantă

Metodele cele mai importante pe care le putem folosi în cazul unei liste sunt enumerate în următorul tabel:

Metoda	Explicații
`public Object` **`getSelectedValue`**`()` **`getCaret`**`()`	Returnează primul element selectat din lista de elemente selectate; returnează `null`, dacă nu este nici un element selectat.

→

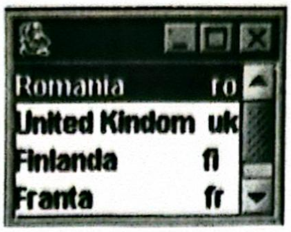
➔

`public void` **`setSelectedValue`** `(Object unObiect, boolean deruleaza)`	Selectează obiectul `unObiect` specificat din listă; dacă `deruleaza` este `true`, atunci se va derula lista pentru a se vizualiza obiectul selectat.
`public void` **`setSelectedIndex`**`(int index)`	Selectează elementul având indicele index.
`public void` **`clearSelection`**`()`	Şterge selecţia curentă.
`public void` **`setVisibleRowCount`** `(int numarCeluleVizibile)`	În cazul în care lista este aşezată într-un `JScrollPane`, această metodă asigură că numai `numarCeluleVizibile` celule vor fi direct vizibile.
`public void` **`ensureIndexIsVisible`** `(int index)`	În cazul în care lista este aşezată într-un `JScrollPane`, apelând această metodă, lista se va derula pentru a face vizibil elementul având indicele `index`.
`public void` **`setSelectionMode`** `(int modSelectie)`	Setează modul de selecţie; `modSelectie` poate fi una dintre constantele: • `ListSelectionModel.SINGLE_SELECTION` - pentru a putea selecta un singur element. • `ListSelectionModel.SINGLE_INTERVALSELECTION` - pentru a putea selecta un singur interval. • `ListSelectionModel.MULTIPLE_INTERVAL_SELECTION` - pentru a putea selecta mai multe intervale.
`public ListModel` **`getModel`**`()` `public void` **`setModel`** `(ListModel model)`	Returnează/setează modelul care ţine datele listei.

Evenimentul pe care îl poate genera o componentă `JList` este `ListSelectionEvent`, care se găseşte în pachetul `javax.swing.event` şi care este aruncat atunci când se selectează unul dintre elementele listei. Prezentăm un exemplu de tratare a acestui eveniment folosind o clasă interioară anonimă. Se va afişa la consolă indicele elementului selectat din listă:

```
lista.addListSelectionListener(new ListSelectionListener()
  {
      public void valueChanged(ListSelectionEvent e)
      {
      int i=lista.getSelectedIndex();
           System.out.println(i);
      }
  });
```

Vom prezenta în continuare un exemplu de utilizare a listelor, în care utilizatorul determină prin apăsarea unei taste selectarea elementului al cărui nume începe cu litera corespunzătoare tastei. Vom folosi, de asemenea, şi clasa `DesenareCelule` care implementează `ListCellRenderer` pentru a formata parţial lista (se schimbă culoarea selecţiei în roşu):



Romania ro
United Kindom uk
Finlanda fi
Franta fr

Fereastră pentru alegerea unei țări

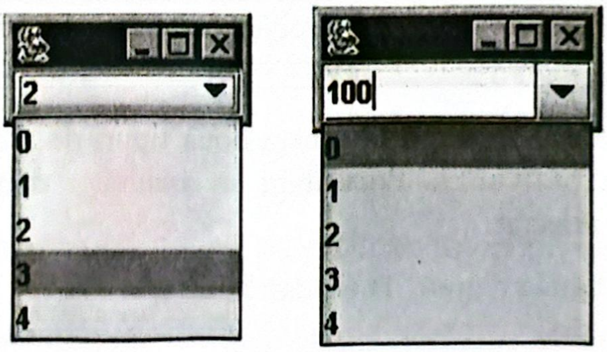
```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class Fereastra extends JFrame {
  // constructor
  public Fereastra () throws Exception {
    String [] state = {
      "Romania                  ro",
      "United Kindom            uk",
      "Finlanda                 fi",
      "Franta                   fr",
      "Grecia                   gr"};

    final JList lista = new JList(state);

    class CautaInLista extends KeyAdapter
    {
      JList lista;
      ListModel model;
      String m_key;
      public CautaInLista(JList list) {
      lista = list;
      model = lista.getModel();
    }

    public void keyTyped(KeyEvent e) {
      char ch = e.getKeyChar();
      if (!Character.isLetterOrDigit(ch))
      return;

      m_key += Character.toUpperrCase(ch);

      for (int k=0; k<model.getSize(); k++) {
         String str = ((String)model.getElementAt(k)).
            toUpperrCase();
            if (str.startsWith(m_key)){
              lista.setSelectedIndex(k);
              lista.ensureIndexIsVisible(k);
              break;
              }
          }
      }
  }

  lista.addKeyListener(new CautaInLista(lista));

    class DesenareCelule extends JLabel implements
     ListCellRenderer {
      public DesenareCelule() {
        setOpaque(true);   }

      public Component getListCellRendererComponent( JList
          list, Object value, int index, boolean isSelected,
          boolean cellHasFocus)
            {
        setText(value.toString());
        setBackground(isSelected ? Color.red : Color.white);
        setForeground(isSelected ? Color.white : Color.black);
        return this; // returneaza chiar aceasta eticheta
      }
    }

    DesenareCelule desen = new DesenareCelule();
    lista.setCellRenderer(desen);
    lista.setVisibleRowCount(4);
    JScrollPane derulare = new JScrollPane(lista);
    this.getContentPane().add(derulare);
    }
}
```

13.6.5. `JComboBox`

Componenta `JComboBox` combină un buton (care se transformă într-un câmp text când componenta este editabilă) cu o listă expandabilă la cerere din care utilizatorul poate să selecteze o singură opţiune la un moment dat. Deoarece componenta `JComboBox` are aceeaşi funcţionalitate ca `JList`, este dificilă alegerea între aceste două posibilităţi. Se recomandă folosirea componentei `JComboBox` în defavoarea `JList`, în cazul în care se doreşte ca utilizatorul să poată să aleagă o singură opţiune pe care să o vadă explicit mai apoi. Când componenta este setată să fie editabilă, atunci va conține un câmp `text` (și nu un buton) în care utilizatorul poate introduce date de intrare. Intern, această componentă este construită dintr-un buton și un meniu `JPopupMenu`, care conține o listă `JList` (reprezentând opțiunile posibile) așezată mai întâi într-un panou `JScrollPane` pentru a se putea derula.

Această componentă permite folosirea mai multor constructori asemănători componentei `JList`. Se poate construi un `JComboBox` folosind un model dat explicit, care este o instanță a clasei `ComboBoxModel` și, în acest caz, vom avea acces maxim asupra proprietăților și comportamentului componentei. Putem să creăm o componentă `JComboBox` direct dintr-un tablou de obiecte `Object` sau dintr-un vector `Vector`. Mai putem crea un `JComboBox` vid, căruia îi putem asocia mai apoi un model de date. Un exemplu de `JComboBox` creat dintr-un tablou găsiți în secțiunea destinată desenării textului. Trebuie să precizăm că avem la dispoziție clasa `DefaultComboBoxModel`, care este implementarea implicită pentru `ComboBoxModel` și pe care o putem folosi drept model efectiv pentru componentele noastre. Mai există un model, și anume interfața `MutableComboBoxModel`, ce face posibile operațiile de adăugare, ștergere și modificare în mod dinamic a conținutului componentei `JComboBox` care o folosește ca model de date.

Prezentăm un exemplu de componentă `JComboBox` construită folosind un model de date care-i gestionează conținutul (același exemplu de la listă modificat pentru `JComponent` – ne putem imagina astfel `JComponent` ca o listă derulantă):



Componenta `JComboBox`

```
import javax.swing.*;
import java.AWT.*;
import java.util.*;
public class Fereastra extends JFrame {
  // constructor
  public Fereastra() throws Exception {
  DefaultComboBoxModel model = new DefaultComboBoxModel() {
           public int getSize() { return 5; }
           public Object getElementAt(int index) { return new
           Integer(index).toString(); }
       };
      JComboBox cb = new JComboBox(model);
/* cb.setEditable(true);
face diferenta intre cele doua capturi editabila/needitabila */
```

```
        this.getContentPane().add(cb);
    }
}
```

Sintetizăm în următorul tabel principalele metode pe care clasa `JComboBox` le pune la dispoziţia dezvoltatorilor, cu remarca că toate metodele care se referă la adăugare, ştergere şi modificare de elemente necesită ca modelul componentei `JComboBox` curente să fie o implementare a clasei `MutableComboBoxModel`:

Metoda	Explicaţie
`public void` **`setEnabled`**`(boolean b)`	Setăm dacă elementul `JComboBox` curent va da posibilitatea editării.
`void` **`addItem`**`(Object o)` `void` **`addItemAt`**`(Object o, int poz)`	Se adaugă obiectul o în lista derulantă, respectiv se inserează înainte de obiectul de pe poziţia `poz`.
`Object` **`getSelectedItem`**`()`	Returnează obiectul curent selectat.

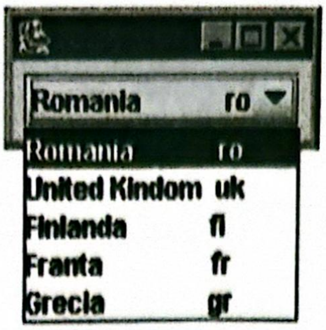
`void` **`removeAllItems`**`()`
`void` **`removeItemAt`**`(int poz)`
`void` **`removeItem`**`(Object ob)`
	Se şterg din lista derulantă toate obiectele, cel de pe poziţia `poz`, respectiv obiectul `ob`.

`int` **`getItemCount`**`()`	Returnează numărul de elemente al meniului derulant.
`ComboBoxModel` **`getModel`**`()` `void` **`setModel`**`(ComboBoxModel)`	Se obţine/setează modelul de date pentru componenta `JComboBox` curentă.

O componentă `JComboBox` poate genera două tipuri de evenimente, şi anume: `ItemEvents` şi `ActionEvents`. Prezentăm un ascultător care afişează la consolă indexul elementului selectat.

```
cb.addItemListener( new ItemListener()
{
  public void itemStateChanged(ItemEvent e)
  {
    System.out.println("Ati ales "+e.getItem().toString());
  }
});
```

Modul în care este vizualizată o componentă `JComboBox` este determinat de o instanţă a clasei `ListCellRenderer` pentru lista derulantă, respectiv `ComboBoxEditor` pentru câmpul `text`. Dacă nu este specificat un alt obiect care să se ocupe de modul de desenare a componentei, ceea ce se va vedea în lista derulantă va fi ceea ce generează `toString()` pentru obiectele componente (ca şi la liste, exceptând obiectele instanţe ale clasei `Icon` care vor fi afişate chiar ca pictograme). Prezentăm, în continuare, un exemplu de personalizare a reprezentării grafice pentru o componentă `JComboBox` împreună cu un exemplu efectiv de folosire a `JComboBox`. Este vorba de exemplul din subcapitolul `JList` remodelat cu `JComboBox`.



Utilizarea unui `ComboBox`

```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class Fereastra3 extends JFrame {
  // constructor
  public Fereastra3 () throws Exception {
      String [] state = {
       "Romania                  ro",
       "United Kindom            uk",
       "Finlanda                 fi",
       "Franta                   fr",
       "Grecia                   gr"};

      final JComboBox cb = new JComboBox(state);

      class DesenareCelule extends JLabel implements
        ListCellRenderer {
        public DesenareCelule() {
               setOpaque(true);   }

      public Component getListCellRendererComponent( JList
      list, Object value, int index, boolean isSelected,
      boolean cellHasFocus)
        {
        setText(value.toString());
        setBackground(isSelected ? Color.red : Color.white);
         setForeground(isSelected ? Color.white : Color.black);
        return this; // returneaza chiar aceasta eticheta
        }
      }

      DesenareCelule desen = new DesenareCelule();
```

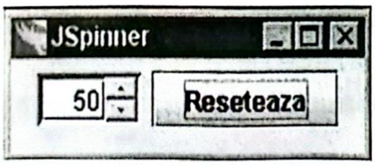
```
        cb.setRenderer(desen);
        this.getContentPane().setLayout(new FlowLayout());
            this.getContentPane().add(cb);
    }
}
```

13.6.6. JSpinner

Reprezintă o singură linie de text care permite utilizatorului să selecteze o valoare dintr-o secvență de valori. `JSpinner` oferă, de obicei, o pereche de săgeți care permite navigarea prin valorile secvenței. Același lucru se poate realiza prin intermediul săgeților sus/jos. Utilizatorul poate să introducă o valoare direct în câmpul text. Componenta `JComboBox` oferă aceleași funcționalități, cu diferența că determină apariția unei liste derulante care prezintă dezavantajul de a se suprapune peste datele importante.

Valoarea zonei de text a componentei `JSpinner` se poate modifica printr-un apel `setValue(Object obiect)` și se poate obține printr-un apel `getValue()`. Componentele `JSpinner` pot arunca evenimente `ChangeEvent` atunci când datele din model se modifică, acestea putând fi captate și tratate de către ascultătorii `ChangeEventListener` atașați componentelor, folosind metoda `addChangeListener (ChangeListener ascultator)`.

Modelul de date pentru această componentă este o clasă care implementează `SpinnerModel`. Există deja trei implementări ale acestei interfețe pe care programatorii le au la dispoziție, și anume: `SpinnerListModel` pentru un tablou de elemente, `SpinnerNumberModel` pentru o listă de numere și `SpinnerDateModel` pentru data calendaristică (pentru dată vom folosi `java.util.Calendar`). Prezentăm în continuare o componentă `JSpinner` construită pe baza unui model `SpinnerNumberModel`.



Componenta `JSpinner`

```
protected void doContinut() throws Exception
   {
   JPanel continut=(JPanel)this.getContentPane();
   continut.setLayout(new FlowLayout());
   Integer valoare = new Integer(50); // valoarea implicita
   Integer minim = new Integer(0); // valoarea minima
   Integer maxim = new Integer(100); // valoarea maxima
   Integer pas = new Integer(1); // pasul de deplasare
   SpinnerNumberModel model = new SpinnerNumberModel
   (valoare, minim, maxim, pas); // apel constructor model
```

```
            final JSpinner spin=new JSpinner(model);
    continut.add(spin,BorderLayout.CENTER);
    JButton but=new JButton("Reseteaza");
    continut.add(but,BorderLayout.EAST);
    but.addActionListener(new ActionListener() {
    public void actionPerformed
    (ActionEvent e) {
    spin.setValue(new Integer(50)); // resetare valoare
       }});
    }
```

13.7. Componente atomice complexe

13.7.1. Componente text

Clasa abstractă `JTextComponent` stă la baza tuturor componentelor text din Swing şi este localizată în pachetul `javax.swing.text`. Toate componentele text se regăsesc în pachetul `javax.swing` şi sunt enumerate în continuare: `JTextField`, `JPasswordField`, `JTextArea`, `JEditorPane`, `JTextPane`.`JTextComponent` – moşteneşte direct clasa `JComponent`. Conţinutul unei componente text este menţinut într-o instanţă a interfeţei `Document`, care reprezintă modelul de date pentru aceasta. Sunt oferite două implementări efective ale interfeţei `Document`: `PlainDocument` şi `Stiled Document`. Prima oferă un singur font, o singură culoare pentru text şi conţinutul format numai din caractere. Cea de a doua este mult mai complexă, permiţând fonturi şi culori multiple, imagini şi componente scufundate în conţinut. `JTextField`, `JPasswordField` şi `JTextArea` au drept model `PlainDocument`, iar `JEditorPane` şi `JTextPane` folosesc ca model `StiledDocument`. Putem obţine sau seta modelul unei componente text folosind metodele `getDocument()`, respectiv `setDocument()`.

Fiecare componentă text are asociat un cursor (engl. *caret*) care marchează poziţia curentă unde se scrie şi care reprezintă o instanţă a clasei `Caret`. Metodele pe care le avem la dispoziţie pentru a gestiona cursorul le prezentăm în tabelul care urmează:

Metoda	Explicaţie
`setCaret``(Caret c)` **`getCaret`**`()`	Permite setarea/obţinerea cursorului asociat componentei text.
`setCaretColor``(Color c)` **`getCaretColor`**`()`	Permite setarea/obţinerea culorii cursorului.
`setCaretPosition``(int pozitie)` **`getCaretPosition`**`()`	Permite setarea/obţinerea poziţiei cursorului în interiorul textului.

Componentele text conţin informaţii despre zonele de text selectate la momentul curent. Putem obţine textul selectat ca `String` folosind `getSelectedText()` şi putem obţine sau seta culoarea de fundal pentru zona selectată folosind metodele `getSelectionBackGround()`, respectiv `setSelectionBackGround()`. De asemenea, sistemul de copiere/lipire (engl. *copy/paste*) ne este accesibil folosind metodele puse la dispoziţie de clasa `JTextComponent`. Programatorii le pot folosi pentru a crea meniuri *pop-up* sau meniuri pentru editare. Descriem în continuare în următorul tabel metodele care sunt puse la dispoziţia utilizatorilor în acest scop:

Metoda	Explicaţie
`copy``()`	Copie zona de text selectată în *clipboard*-ul sistemului de operare folosit.
`cut``()`	Mută zona de text selectată în *clipboard*-ul sistemului de operare folosit.
`paste``()`	Lipeşte la poziţia curentă a cursorului ceea ce se găseşte în *clipboard* la momentul curent.
`select``(int pozIn, int pozFin)`	Selectează textul componentei text începând cu poziţia de început `pozIn` şi terminând cu poziţia de sfârşit `pozFin`.
`selectAll``()`	Selectează întreg textul componentei.

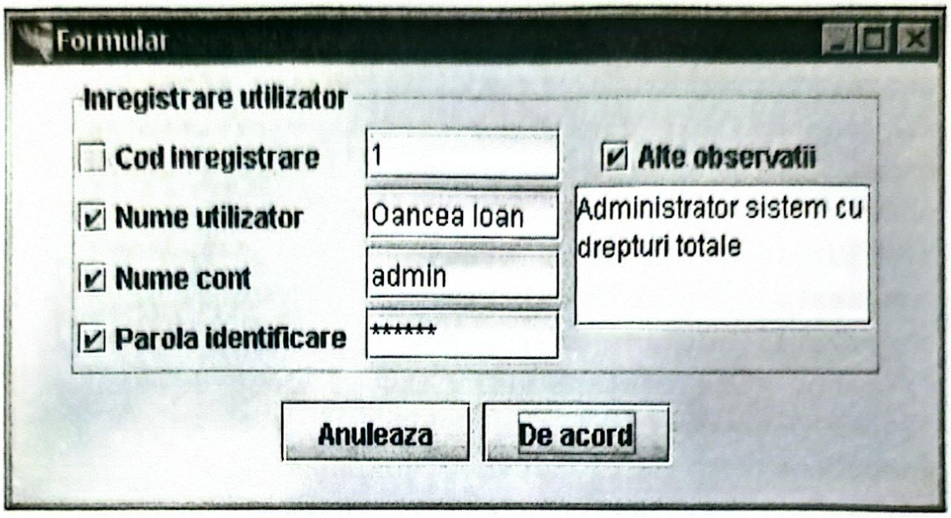
Un exemplu de folosire a acestor metode îl veţi găsi în subcapitolul referitor la meniuri *pop-up*. Folosind combinaţiile de taste (acestea sunt dependente de *look-and-feel*-ul folosit) *Ctrl+C*, respectiv *Ctrl+V* putem realiza *copy/paste* fără a scrie cod pentru aceasta. De asemenea, putem selecta text folosind *Shift* şi tastele cursor.

Sunt puse la dispoziţia programatorilor diverse metode prin care se permite manevrarea conţinutului zonei de text. Astfel, conţinutul unei componente text se poate seta sau obţine folosind metoda `setText()`, respectiv `getText()`. Conţinutul se poate, de asemenea, trimite spre un flux de ieşire folosind metoda `write(Writer iesire)` sau se poate folosi metoda `read(Reader intrare, Object descriere)` pentru a crea un model `Document` nou al cărui conţinut este dat de intrare şi care se va ataşa componentei text, vechiul model fiind pierdut.

Prezentăm, în continuare, un exemplu care ilustrează cele trei componente text simple, şi anume `JTextField`, `JTextArea` şi `JPasswordField`. Vom pune în evidenţă modul în care modelul de date ne ajută în crearea de componente personalizate. Extinzând modelele care stau la baza componentelor text, putem crea zone de text care să permită doar scrierea literelor sau numai a caracterelor, zone de text având un număr fix de caractere etc. Panoul care urmează reprezintă un formular ce va trebui completat de către utilizatorii aplicaţiei, conţinând:

- o zonă text `CampNumar` care extinde `JTextField`, oferind posibilitatea introducerii de text conţinând numai cifre şi având o dimensiune precizată.
- un câmp `JTextField` obişnuit care nu are nici o constrângere impusă.
- un câmp `JTextFieldPersonalizat` care permite doar introducerea unui număr precizat de caractere, ce trebuie să fie litere sau spaţiu.
- un câmp `JPasswordField` care reprezintă o parolă ce se aşteaptă a fi introdusă.
- un câmp `Continut` care extinde `JTextArea` oferind un număr constant de caractere şi determinând pierderea focusului componentei la apăsarea tastei `Tab`.

Butonul `De acord` determină realizarea unei anumite acţiuni folosind conţinutul colectat din zonele de text, folosind metoda `getText()`, iar butonul `Anuleaza` determină ştergerea conţinutului tuturor componentelor text folosind apeluri de genul `setText("")`. Am folosit, de asemenea, butoane `CheckBox` pentru a permite utilizatorului actualizarea facilă a eventualei baze de date care stă în spatele formularului prin excluderea anumitor câmpuri determinate de butoane neselectate. Poziţionarea componentelor în container s-a făcut folosind gestionarul `GridLayout`.



☐ Cod inregistrare: 1

☑ Nume utilizator: Oancea Ioan

☑ Nume cont: admin

☑ Parola identificare: ******

☑ Alte observatii: Administrator sistem cu drepturi totale

Formular pentru înregistrarea utilizatorilor

```
void doFormular(JFrame f)
{
     JPanel intrari = new JPanel();
     intrari.setBorder(
       new TitledBorder("Inregistrare utilizator"));
     // panou stang care tine checkbox-urile
     JPanel intraris = new JPanel();
     intraris.setLayout(new GridLayout(0, 1, 2, 1));
     JCheckBox numarCurent =
       new JCheckBox("Cod inregistrare");
     JCheckBox numeUtilizator =
       new JCheckBox("Nume utilizator");
     JCheckBox numeCont = new JCheckBox("Nume cont");
     JCheckBox parola = new JCheckBox("Parola identificare");
     numeUtilizator.setSelected(true);
     numeCont.setSelected(true);
     intraris.add(numarCurent);
     intraris.add(numeUtilizator);
     intraris.add(numeCont);
     intraris.add(parola);
     // panou drept care stocheaza zonele de text
     JPanel intrarid = new JPanel();
     intrarid.setLayout(new GridLayout(0, 1, 5, 3));
     final CampNumar textNumarCurent=new CampNumar(4,5);
     textNumarCurent.setFocusable(true);
```

```
final JTextField textNumeUtilizator=new JTextField(8);
final JTextFieldPersonalizat textNumeCont=
new JTextFieldPersonalizat(8,6);
final JPasswordField textParola=new JPasswordField(8);
intrarid.add(textNumarCurent);
intrarid.add(textNumeUtilizator);
intrarid.add(textNumeCont);
intrarid.add(textParola);
// zona de text JTextArea pentru observatii
final Continut observatii=new Continut(2,5);
observatii.setWrapStyleWord(true);
observatii.setLineWrap(true);
observatii.setTabSize(222);
observatii.setMaximumSize(new Dimension(50,50));
// panou care contine eticheta si zona de text Continut
JPanel panouContinut=new JPanel();
panouContinut.setLayout(new BoxLayout(panouContinut,
BoxLayout.Y_AXIS));
final JCheckBox Continut=
  new JCheckBox("Alte observatii");
Continut.setSelected(true);
panouContinut.add(Continut);
JScrollPane sp=new JScrollPane();
sp.setViewportView(observatii);
sp.setHorizontalScrollBarPolicy(JScrollPane.
  HORIZONTAL_SCROLLBAR_NEVER );
panouContinut.add(sp);
panouContinut.add(new JLabel("       "));
panouContinut.setMaximumSize(new Dimension(20,20));

JPanel tot= new JPanel();
tot.setLayout(new BoxLayout(tot, BoxLayout.X_AXIS));
tot.add(intraris);
tot.add(intrarid);

intrari.setLayout(new BorderLayout());
intrari.add(tot, BorderLayout.WEST);
intrari.add(new JLabel("  "), BorderLayout.CENTER);
intrari.add(panouContinut, BorderLayout.EAST);
// constructie butoane
JPanel butoane = new JPanel();
butoane.setLayout (new FlowLayout(FlowLayout.LEFT));
JButton anuleaza = new JButton("Anuleaza");
JButton deacord = new JButton("De acord");
   butoane.add(anuleaza);
```

```
        butoane.add(deacord);
        //ascultatori Butoane
        anuleaza.addActionListener(new ActionListener() {
             public void actionPerformed(ActionEvent e)
        {
        textNumarCurent.setText("");
        textNumeUtilizator.setText("");
        textNumeCont.setText("");
        textParola.setText("");
        observatii.setText("");
        }});
        deacord.addActionListener(new ActionListener() {
             public void actionPerformed(ActionEvent e)
             {
             String NumarCurent=textNumarCurent.getText();
             String NumeUtilizator=textNumeUtilizator.
             getText();
             String NumeCont=textNumeCont.getText();
             char[] parola = textParola.getPassword();
             String obs=observatii.getText();
             // eventuale actiuni ale utilizatorilor
             }});
        //panou mare
        JPanel jsp=new JPanel();
        jsp.add(intrari);
        jsp.add(butoane);
        f.getContentPane().add(jsp,BorderLayout.CENTER);
        f.setVisible(true);
    }
```

În procesul de personalizare a componentelor text, suprascriem pentru acestea metoda având prototipul `protected Document createDefaultModel()`, care permite adăugarea drept model de date implicit, în procesul de creare a componentei, a propriului nostru model, ce reprezintă, de fapt, o implementare a clasei `PlainDocument`. Singura metodă din această clasă care trebuie suprascrisă este `public void insertString(int deplasament, String sir, AttributeSet a) throws BadLocationException`, care este apelată de componentă de fiecare dată când un caracter sau un şir (spre exemplu, când lipim text componentei) este inserat în document.

Prezentăm clasa `JTextFieldPersonalizat`, care este utilizată în exemplul precedent pentru obţinerea de componente text având un număr fix de caractere care trebuie să fie litere sau caracterul spaţiu. Aceste condiţii sunt verificate folosind metodele `Character.isLetter(char c)` şi `Character.isSpaceChar (char c)`, iar dacă acestea nu se îndeplinesc, se va emite un sunet de avertisment. Pentru construirea unei componente de acest tip se va folosi constructorul `public JTextFieldPersonalizat(int l, int init)`, unde `l` reprezintă lungimea zonei de text, iar `init` reprezintă numărul maxim de caractere.

```
// pentru a restrictiona numarul de caractere introduse
import javax.swing.*;
import javax.swing.text.*;
import java.awt.Toolkit;
import java.util.Locale;

public class JTextFieldPersonalizat extends JTextField {
   private Toolkit toolkit;
   int nrCaractere=0;
   public JTextFieldPersonalizat(int l,int init)
   {
      super(init);
      nrCaractere=l;
      toolkit = Toolkit.getDefaultToolkit();
   }

   public JTextFieldPersonalizat(int l)
   {
      super();
      nrCaractere=l;
      toolkit = Toolkit.getDefaultToolkit();
   }

   protected Document createDefaultModel() {
      return new DocumentPersonalizat();
   }

  protected class DocumentPersonalizat extends PlainDocument {
    public void insertString(int deplasament,  String sir,
     AttributeSet a)  throws BadLocationException
   {
       char[] sursa = sir.toCharArray();
       char[] rezultat = new char[sursa.length];
       int j = 0;
       if(JTextFieldPersonalizat.this.getText().
          length()<nrCaractere)
       for (int i = 0; i < rezultat.length; i++)
          {
          if (deplasament<=nrCaractere)
            if (Character.isLetter(sursa[i]) ||
               Character.isSpaceChar(sursa[i]) )
                   { rezultat[j++] = sursa[i]; }
            else {      toolkit.beep();      }
          else { toolkit.beep();  }
```

```
              super.insertString(deplasament, new String(
                rezultat, 0, j), a);
              }
    }
}
```

Pentru clasa `CampNumar`, care permite introducerea unui număr fix de cifre, diferența față de `JTextFieldPersonalizat` constă în verificarea condiției `Character.isDigit(char c)` care returnează `true` în cazul în care `c` este o cifră.

În cazul `JTextArea`, prezentăm în continuare implementarea unui model personalizat care va determina transferarea focusului în momentul apăsării tastei `Tab` (acest lucru nu se petrece, fiind implicit un incovenient ce trebuie evitat), permițând și introducerea unui număr limitat de caractere:

```
protected class DocumentPersonalizat extends PlainDocument {
    public void insertString(int deplasament,  String sir,
      AttributeSet a)   throws BadLocationException
      {     char[] sursa = sir.toCharArray();
           char[] rezultat = new char[sursa.length];
           int j = 0;
           if(Continut.this.getText().length()<numarCaractere)
           for (int i = 0; i < rezultat.length; i++)
           {
           if (deplasament<=numarCaractere)
                 if ( Character.isLetterOrDigit(sursa[i]) ||
                    Character.isSpaceChar(sursa[i]) )
                    {rezultat[j++] = sursa[i];
                    }
                 else
                      { if (sursa[i]=='\t')
                           Continut.this.transferFocus(); }
                        else {      toolkit.beep();    }
                      }
           super.insertString(deplasament, new String
                      (rezultat,0, j), a);
           }
      }
```

Considerăm `Continut` numele componentei care extinde `JTextField` și `numarCaractere`, numărul de caractere maxim care poate fi introdus. Transferul focusului se face prin intermediul metodei `transferFocus()` pe care orice componentă grafică o moștenește din clasa `Component`.

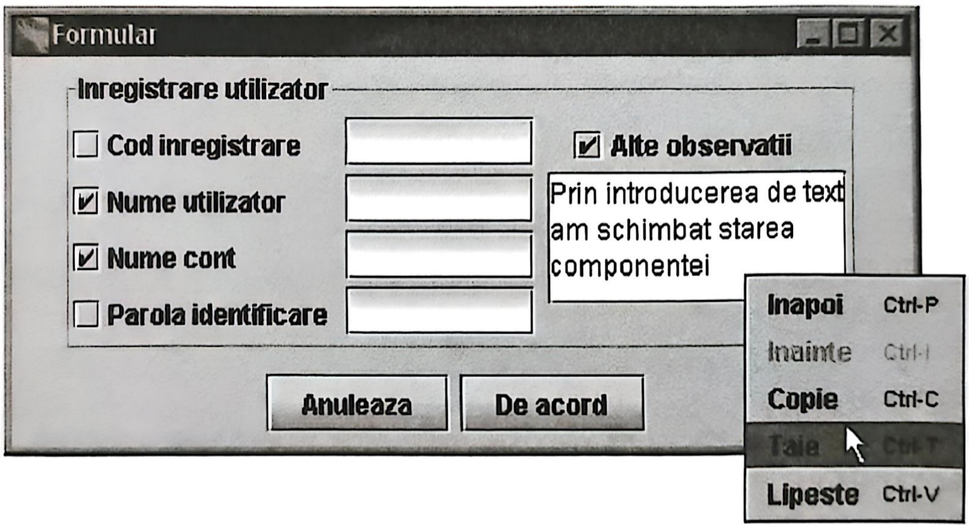
În cazul componentei `JPasswordField`, citirea conținutului nu se face cu `getText()`, o astfel de încercare determinând un avertisment (engl. *warning*), deoarece această metodă a devenit învechită. Pentru a obține conținutul unei componente `JPasswordField` se folosește metoda `getPassword()` ce returnează un tablou cu elemente `char`, care eventual poate fi criptat înainte de a fi stocat într-un fişier sau o bază de date, folosind API-urile pentru criptografie cu criptosistemele pe care Java le implementează (a se vedea pachetul `javax.cripto` şi pachetele dependente de acesta).

Swing pune la dispoziţia implementatorilor un sistem standard pentru a răspunde la dorinţele utilizatorilor aplicaţiei de a-şi anula o anumită acţiune greşită (engl. *undo*), respectiv navigarea prin mulţimea stărilor mai vechi ale aplicaţiei determinate de acţiunile utilizatorilor (engl. *undo/redo system*). Acest mecanism este implementat prin construcţia componentelor text. Tot ceea ce trebuie să facem este să adăugam componentei text, folosind metoda `addUndoableEditListener()`, un ascultător al evenimentelor `UndouableEditEvent`, care sunt aruncate atunci când starea componentei se modifică (spre exemplu, s-a introdus sau s-a şters text). Ascultătorul foloseşte un gestionar – o instanţă a clasei `UndoManager` –, în care adaugă acţiunile încapsulate în evenimente. Gestionarul permite, mai apoi, prin intermediul metodelor `undo()`, `redo()` accesul la aceste acţiuni realizate de utilizatori şi, deci parcurgerea secvenţei de stări ale aplicaţiei.

Exemplul care urmează ataşează un meniu *pop-up* componentei text `observatii` din exemplul precedent, care se declanşează la apăsarea butonului drept al mouse-ului (pentru aceasta am adăugat un ascultător `MouseListener`). Meniul *pop-up* permite realizarea operaţiilor de *undo/redo*, precum şi *copy/cut/paste*. Pentru amănunte puteţi urmări codul comentat, care este o completare la exemplul precedent:

```
import javax.swing.undo.*;
//...
final Continut observatii=new Continut(2,5);
//...
final JPopupMenu pop=new JPopupMenu("Operatii");
final UndoManager gestionar=new UndoManager();
final JMenuItem si=new JMenuItem("Inapoi");
pop.add(si);
si.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_P,
    ActionEvent.CTRL_MASK));
si.setEnabled(false);
final JMenuItem su=new JMenuItem("Inainte");
pop.add(su);
su.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_I,
    ActionEvent.CTRL_MASK));
su.setEnabled(false);
si.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      try { gestionar.undo(); }
      catch (CannotRedoException cre) { cre.printStackTrace(); }
      si.setEnabled(gestionar.canUndo());
    su.setEnabled(gestionar.canRedo());
    }
});

su.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      try { gestionar.redo(); }
      catch (CannotRedoException cre) { cre.printStackTrace(); }
           si.setEnabled(gestionar.canUndo());
           su.setEnabled(gestionar.canRedo());
           }
});

observatii.getDocument().addUndoableEditListener(new
  UndoableEditListener() {
    public void undoableEditHappened(UndoableEditEvent e) {
      gestionar.addEdit(e.getEdit());
      si.setEnabled(gestionar.canUndo());
      su.setEnabled(gestionar.canRedo());
      }
});

JMenuItem copie=new JMenuItem("Copie");
copie.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      observatii.copy();
      }}
);
pop.add(copie);
copie.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_C,
    ActionEvent.CTRL_MASK));

JMenuItem taie=new JMenuItem("Taie");
taie.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
    observatii.cut();
    }});
pop.add(taie);
taie.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_T,
    ActionEvent.CTRL_MASK));

JMenuItem lipeste=new JMenuItem("Lipeste");
lipeste.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
```

```
    observatii.paste();
    }});
pop.add(lipeste);
lipeste.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_V,
    ActionEvent.CTRL_MASK));

observatii.addMouseListener(new MouseAdapter()
  {
    public void mousePressed(MouseEvent e)
    {
    int i=e.getClickCount();
    if (e.getButton()==3 && i==1)
    {
    pop.show(e.getComponent(), e.getX(), e.getY());
    pop.setVisible(true);
    }
  }
});
//...
```


Formular
Inregistrare utilizator
☐ Cod inregistrare
☑ Nume utilizator
☑ Nume cont
☐ Parola identificare
☑ Alte observatii
Anuleaza
De acord
Inapoi Ctrl-P
Inainte Ctrl-I
Copie Ctrl-C
Taie Ctrl-T
Lipeste Ctrl-V

Prin introducerea de text am schimbat starea componentei

Meniu *pop-up* ataşat componentei text

Pe lângă aceste componente text relativ simple, programatorul mai are la dispoziţie şi altele mult mai complexe, şi anume cele care au drept model `StiledDocument` prin care se oferă suport pentru formatele RTF sau HTML.

13.7.2. `JTable`

`JTable` este componenta grafică care permite operarea cu tabele. Clasa `JTable` din pachetul `javax.swing` extinde direct clasa `JComponent`, implementând interfeţele `TableModelListener`, `TableColumnModelListener`, `ListSelectonListener`, `CellEditorListener` şi `Scrollable`. `JTable` are asociate, de asemenea, trei modele: `TableModel`, `TableColumnModel` și `ListSelectionModel`, fiecare gestionând o anumită caracteristică a acestei componente. De asemenea, datorită complexității acestei componente, `JTable` are un întreg pachet dedicat, și anume `javax.swing.table`.

Componenta `JTable` este construită pe același principiu clasic în Swing, numit model-delegat (engl. *model-delegate*), care reprezintă, de fapt, o despărțire a datelor de modul lor de vizualizare. Datele tabelei vor fi întotdeauna ținute într-o matrice, într-o listă de vectori, sau o listă de liste, de fapt, orice structură de date tabelară. Reprezentarea grafică a datelor (engl. *look*) este încapsulată împreună cu modul în care componenta reacționează la acțiunile utilizatorilor (engl. *feel*) într-un obiect, care poartă denumirea *delegate*, asociat fiecărei componente grafice. Componenta joacă astfel rolul unui mediator între vizualizare și datele care reprezintă conținutul ce se vrea vizualizat. Pentru `JTable` obiectul *delegate* implicit are numele `TableUI`, se găsește în pachetul `javax.swing.plaf` și poate fi schimbat prin adăugarea unui *plug-in* nou pentru schimbarea vizualizării și comportării la acțiunile utilizatorilor (a se vedea subcapitolul 13.2.5.3). Această clasă dă, cu alte cuvinte, pentru componenta `JTable` vizualizarea și comportamentul (engl. *look-and-feel*). Datele ce se doresc vizualizate sunt la rândul lor înfășurate într-un model care prin sistemul de ascultători actualizează automat vizualizarea în urma modificărilor făcute în structura de date. Spre exemplu, atunci când vrem să ștergem o înregistrare dintr-o tabelă, o ștergem din structura care ține datele și atenționăm componenta `JTable` care o înfășoară să se reactualizeze.

Prezentăm în continuare cele trei modele ale unei instanțe `JTable`, împreună cu descrierile corespunzătoare:

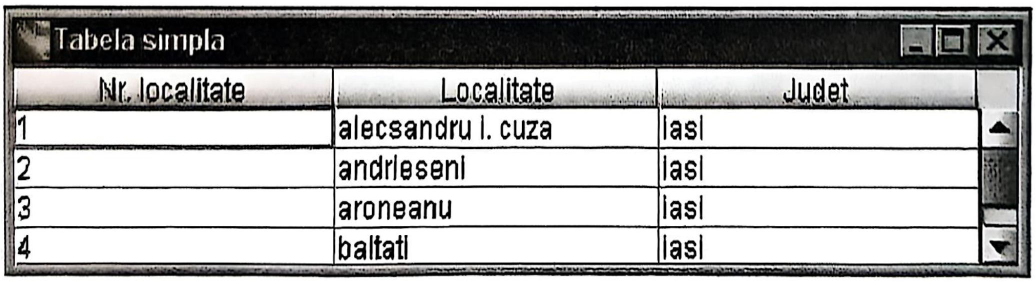
Model	Funcționalitate
`TableModel`	Modelul care înfășoară datele ce se vor vizualizate într-o tabelă este întotdeauna gestionat printr-o clasă care implementează interfața `TableModel`. Implementările acestei interfețe specifică structura de date care se vrea vizualizată printr-o tabelă, precum și modalitățile de mânuire a acestei structuri.
`TableColumnModel`	Este folosit pentru a gestiona instanțe ale clasei `TableColumn` care reprezintă coloane ale tabelelor `JTable`, oferind control asupra ordinii, selectării coloanelor și mărimii marginilor.
`ListSelectionModel`	Este folosit pentru a oferi mai multe moduri de selecție, și anume: un singur element, un singur interval și mai multe intervale.

Clasa `TableColumn` este responsabilă pentru vizualizarea unei coloane din tabela curentă. Fiecare instanță `TableColumn` are asociată câte o clasă care gestionează desenarea celulelor coloanei, editarea celulelor coloanei, desenarea capului de coloană, acestea permițând personalizarea fiecărei coloane în parte. Dacă aceste clase asociate coloanelor sunt setate `null`, se va folosi mecanismul implicit de desenare și editare atașat tabelei în întregime. Clasele care gestionează desenarea celulelor sunt instanțe ale interfeței `TableCellRenderer`, iar cele care gestionează editarea celulelor sunt instanțe ale clasei `CellEditor`. Pentru fiecare coloană putem specifica modalitatea în care ea reacționează la redimensionarea celorlalte coloane, când apar liniile despărțitoare între coloane şi când nu, dimensiunea marginilor dintre linii şi coloane, culoarea de fundal pentru celulele selectate, înălţimea celulelor, precum şi lăţimea coloanelor.

`JTable` oferă mai mulţi constructori pentru realizarea tabelelor, pe care îi vom sintetiza în tabelul următor datorită diversităţii lor:

Constructor	Explicaţie
`JTable``()`	Construieşte un tabel care este iniţializat cu modelul de date implicit, modelul implicit pentru coloane şi modelul implicit pentru selecţie.
`JTable``(int nrLinii, int nrColoane)`	Construieşte un tabel cu dimensiunile `nrLinii` şi `nrColoane` având celulele goale şi drept model de date, `DefaultTableModel`.
`JTable``(Object[][] liniiDate, Object[] numeColoane)`	Construieşte un tablou pentru vizualizarea unei matrice, `liniiDate` având numele coloanelor `numeColoane`.
`JTable``(TableModel dm)`	Construieşte un tabel având drept model de date `dm`, modelul implicit pentru coloane şi modelul implicit pentru selecţie.
`JTable``(TableModel dm, TableColumnModel cm)`	Construieşte un tabel care este iniţializat cu modelul de date `dm`, modelul pentru coloane `cm` şi modelul implicit pentru selecţie.
`JTable``(TableModel dm, TableColumnModel cm, ListSelectionModel sm)`	Construieşte un tabel care este iniţializat cu modelul de date `dm`, modelul pentru coloane `cm` şi modelul `sm` pentru selecţie.
`JTable``(Vector liniiDate, Vector numeColoane)`	Construieşte un tablou pentru vizualizarea unui `Vector` de elemente Vector `liniiDate` având numele coloanelor într-un Vector `numeColoane`.

Prezentăm în continuare un exemplu de tabelă simplă având toate modelele implicite, construită folosind unul dintre constructorii enumeraţi:



Nr. localitate	Localitate	Judet
1	alecsandru i. cuza	iasi
2	andrieseni	iasi
3	aroneanu	iasi
4	baltati	iasi

Componenta `JTable`

Am folosit metoda care urmează:

```
void creeazaTabela(JFrame f)
{
Object[][] liniiDate = {
 {"1", "alecsandru i. cuza", "iasi"},
 {"2", "andrieseni", "iasi"},
 {"3", "aroneanu", "iasi"},
    {"4", "baltati", "iasi"},
    {"5", "belcesti", "iasi"}
  };
  String[] numeColoane = {"Nr. localitate", "Localitate", "Judet"};
  JTable tabela = new JTable(liniiDate, numeColoane);
  JScrollPane jsp = new JScrollPane(tabela);
  f.getContentPane().add(jsp,BorderLayout.CENTER);
  f.setVisible(true);
  }
```

Chiar dacă nu am creat explicit modele pentru date, coloane şi selecţie, acestea au fost create implicit, aşa cum se poate vedea şi din explicaţia constructorului folosit. De exemplu, ca model de date, i s-a ataşat tabelei implicit o instanţă a clasei `Default TableModel` la care avem acces prin metoda `getModel()` din clasa `JTable`. Dacă dorim să ataşăm un model unei componente `JTable`, putem folosi metoda `setModel()`.

Ne vom opri, în continuare, la modelul de date asociat unei componente `JTable`. După cum se observă din lista de constructori ai clasei `JTable`, în general, mai întâi se construieşte modelul de date (modele, în general) şi mai apoi acesta este folosit în construcţia efectivă a tabelei. După cum am mai spus, interfaţa `TableModel` stă la baza modelelor de date pentru o tabelă şi conţine nouă metode care trebuie implementate de programatori, pe care le prezentăm în continuare:

Metodele interfeţei	Explicaţia
`public void` **`addTableModelListener`** `(TableModelListener l)`	Adaugă un ascultător în lista de ascultători care va fi anunţat de fiecare dată când apare un eveniment model.
`public Class` **`getColumnClass`** `(int indiceColoana)`	Clasa cea mai generală a elementelor celulelor coloanei având indicele `indiceColoana`.
`public int` **`getColumnCount`**`()`	Returnează numărul de coloane.
`public String` **`getColumnName`** `(int indiceColoana)`	Returnează numele coloanei având indicele `indiceColoana`.
`public int` **`getRowCount`**`()`	Returnează numărul de linii.
`public Object` **`getValueAt`** `(int indiceLinie, int indice Coloana)`	Returnează elementul de la linia `indiceLinie` şi coloana `indiceColoana`.
`public boolean` **`isCellEditable`** `(int indiceLinie, int indiceColoana)`	Returnează `true` dacă celula de la linia `indiceLinie` şi coloana `indiceColoana` se vrea editabilă.
`public void` **`removeTableModelListener`** `(TableModelListener l)`	Şterge un ascultător din lista de ascultători care captează evenimentele modelului `TableModelEvent`.
`public void` **`setValueAt`** `(Object valoare, int indiceLinie, int indiceColoana)`	Setează valoarea `valoare` pentru linia `indiceLinie` şi coloana `indiceColoana`.

Acestea specifică modul de lucru cu structura de date care va da conţinutul tabelei şi vor fi folosite în mod direct de componenta `JTable` în procesul de desenare a tabelei. De exemplu, se ştie că în momentul desenării componenta `JTable` apelează foarte des metoda `getRowCount()` şi programatorii trebuie să o implementeze eficient. Demonstrativ, vom rescrie metoda care desenează o tabelă într-o fereastră, din exemplul precedent, folosind de data aceasta un model personalizat. Drept rezultat se va obţine aceeaşi vizualizare. Singura diferenţă este că tabela nu mai este editabilă, datorită implementării pe care o vom da metodei `isCellEditable()`.

```
void creeazaTabela(JFrame f)
{

class ModelulMeu implements TableModel
{
Object[][] liniiDate = {
  {"1", "alecsandru i. cuza", "iasi"},
  {"2", "andrieseni", "iasi"},
  {"3", "aroneanu", "iasi"},
  {"4", "baltati", "iasi"},
  {"5", "belcesti", "iasi"}
};
String[] numeColoane = {"Nr. localitate", "Localitate", "Judet"};
public void addTableModelListener(TableModelListener l)
{ /* nu dam implementare acestei metode
   */       }
public Class getColumnClass(int indiceColoana)
{ return liniiDate[1][indiceColoana].getClass();
/* in fapt String */            }
public int getColumnCount()
{ return numeColoane.length;            }
public String getColumnName(int indiceColoana)
{  return numeColoane[indiceColoana];           }
public int getRowCount()
{ return liniiDate.length;           }
public Object getValueAt(int indiceLinie,int indiceColoana)
{ return liniiDate[indiceLinie][indiceColoana];          }
public boolean isCellEditable(int indiceLinie,int indiceColoana)
{ return false;             }
public void removeTableModelListener(TableModelListener l)
{ //nu dam implementarea acestei metode             }
public void setValueAt(Object Valoare, int indiceLinie,int
indiceColoana)
{
 /* nu dam implementarea; folosind aceasta metoda se poate
```

```
     reactualiza continutul modelului (structurii de date) dupa
     ce introducem date in celulele vizualizarii */
   }
   }
   ModelulMeu m = new ModelulMeu();
   JTable tabela = new JTable(m);
   JScrollPane jsp = new JScrollPane(tabela);
   f.getContentPane().add(jsp,BorderLayout.CENTER);
   f.setVisible(true);
   }
```

Swing pune la dispoziţie două clase care reprezintă implementări implicite (engl. *default*) ale interfeţei `TreeModel` şi care pot fi folosite, de asemenea, prin extindere pentru crearea modelelor de date proprii. Aceste clase sunt `AbstractTableModel` şi `DefaultTableModel`. Prima este preferabil a fi extinsă, deoarece oferă libertate în alegerea structurii de date care stă la baza modelului. În cazul clasei `DefaultTableModel`, se ştie că structura de date este un `Vector` (a se vedea clasa `Vector`) de elemente `Vector`.

În cazul în care extindem clasa `AbstractTableModel`, tot ceea ce trebuie să facem este să suprascriem trei metode care încapsulează structura aleasă pentru a ţine datele tabelei într-un model. Cele trei metode care trebuie suprascrise sunt:

Prototipul metodei	Explicaţia metodei
`public int` **`getRowCount`**`()`	Folosită de componenta `JTabel` corespunzătoare pentru a afla numărul de linii.
`public int` **`getColumnCount`**`()`	Folosită de componenta `JTabel` corespunzătoare pentru a afla numărul de coloane.
`public Object` **`getValueAt`** `(int row, int column)`	Folosită de componenta `JTabel` corespunzătoare pentru a afla conţinutul celulelor pe care trebuie să-l vizualizeze.

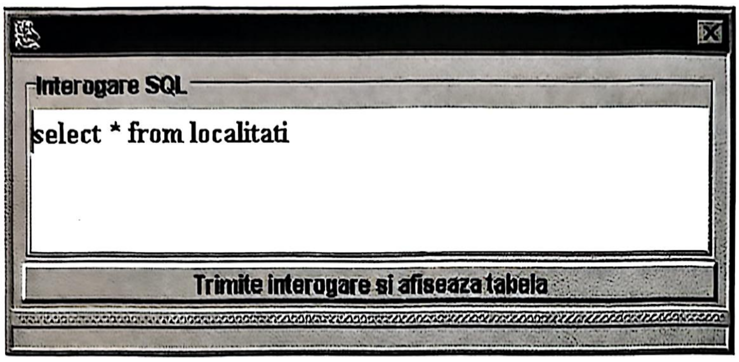
Un model poate fi folosit ca un adaptor între `JTable` şi structura de date pe care o vizualizează. Ca exemplu pentru clasa `AbstractTableModel`, prezentăm în continuare clasa `JDBCAdaptor` care reprezintă un adaptor pentru o tabelă stocată pe un server de baze de date, încapsulând rezultatul obţinut în urma interogărilor efectuate.

Amănunte referitoare la baze de date şi SQL puteţi găsi în capitolul 8, dedicat API-ului JDBC. Urmează câteva explicaţii asupra codului prezentat mai jos. Prin constructor se realizează conexiunea la tabela ce se doreşte vizualizată. Folosind metoda `trimiteInterogare(String interogare)` vom stoca într-un vector de vectori datele tabelei, şi anume rezultatul interogării păstrat într-un obiect `ResultSet`. Folosim clasa pe care Java o pune la dispoziţie pentru a implementa vectori, şi anume clasa `Vector` care reprezintă un vector sau elemente `Object`, deci şi obiecte `Vector`. Obiectul `linii` va memora în final datele tabelei. De asemenea, în tabloul `numeColoane` obţinem numele câmpurilor din tabelă, acest lucru ajutându-ne la atribuirea de denumiri sugestive coloanelor din tabela vizualizată. Pentru aceasta suprascriem, în continuare, metoda `getColumnName()` din interfaţa `Model`. Dacă nu am fi făcut această suprascriere, capul de tabel la afişare ar fi conţinut numele câmpurilor din tabela MySql (spre exemplu, `nr_loc` în loc de `Nr. localitate` aşa cum va apărea după suprascriere). Urmează, mai apoi, cele trei metode ale clasei `AbstractTableModel`, despre care am vorbit mai sus, a căror implementare este extrem de simplă.

```
// aceasta clasa incapsuleaza tabela interogata
// (Model)
import java.util.Vector;
import java.sql.*;
import javax.swing.table.AbstractTableModel;
import javax.swing.event.TableModelEvent;
import javax.swing.*;
import java.awt.*;

public class JDBCAdaptor extends AbstractTableModel
    {// variabile legate de SQL
    static Connection conexiune;
    static Statement stare;
    static ResultSet rezultat;
    static ResultSetMetaData meta;
    //variabile in care tinem datele tabelei
    String[] numeColoane = {};
    Vector linii = new Vector();
    //variabile legate de actualizare
    static boolean facemActualizare=false;

    static boolean editabila;
    //constructorul adaptorului
    public JDBCAdaptor(String bazaDate) {
    //stabilesc conexiunea cu baza de date
    try {

        Class.forName("org.gjt.mm.mysql.Driver").newInstance();
        String url="jdbc:mysql://localhost/"+bazaDate;
        conexiune = DriverManager.getConnection(url,"","");
        stare = conexiune.createStatement();        }
    catch(Exception ex) {
    System.err.println("Eroare la conectare");
    System.err.println(ex);        }
        }
    //execut interogarea "intrebare" catre tabela curenta si
    //culeg rezultatul in vectorul de linii
    public void trimiteInterogare(String intrebare) {
     if (conexiune == null || stare == null) {
         System.err.println(
         "Nu avem asupra cui sa facem interogarea");
         return;          }
     try {
         rezultat = stare.executeQuery(intrebare);
         meta = rezultat.getMetaData();
         int numarColoane =  meta.getColumnCount();
         numeColoane = new String[numarColoane];
         // obtinem numele coloanelor
         for(int c = 0; c < numarColoane; c++) {
             numeColoane[c] = meta.getColumnLabel(c+1);
         }
         /* luam liniile tabelei de pe server si le
            introducem in tabloul de vectori */
         linii = new Vector();
         int incep=1;
         while (rezultat.next()) {
           incep++;
           Vector l = new Vector();
           for (int i = 1; i <= getColumnCount(); i++) {
               l.addElement(rezultat.getObject(i));
           }
           linii.addElement(l);
           }
         }
   catch (SQLException ex) {System.err.println(ex);        }
}

public String getColumnName(int column)
{    String c="";
     if (numeColoane[column] != null)
        if (numeColoane[column].indexOf("nr_loc")!=1)
             c="Nr. localitate";
        else if (numeColoane[column].indexOf("localitate")!=1)
               c="Localitate";
              else if (numeColoane[column].indexOf("plasa")!=1)
               c="Plasa";
              else
              if (numeColoane[column].indexOf("judet")!=1)
               c="Judet ";
     return c;      }

 //   Implementarea efectiva a adaptorului
public int getColumnCount() {
```

```
            return numeColoane.length;   //numar coloane
        }
        public int getRowCount() {
            return linii.size();    //numar linii
        }
        public Object getValueAt(int linie, int coloana) {
            Vector row = (Vector)linii.elementAt(linie);
            return row.elementAt(coloana);
            //continutul celulelor
        }
}
```

Odată modelul creat, printr-o metodă `public void setModel(TableModel dataModel)`, acesta se poate asocia unui obiect `JTable`. O altă metodă este folosirea constructorului de forma `public JTable(TableModel dm)` prin care se creează o reprezentare vizuală a unei tabele, având drept conţinut datele din model şi care se poate mai apoi introduce într-o componentă `JScrollPane` pentru a se putea derula.



select * from localitati

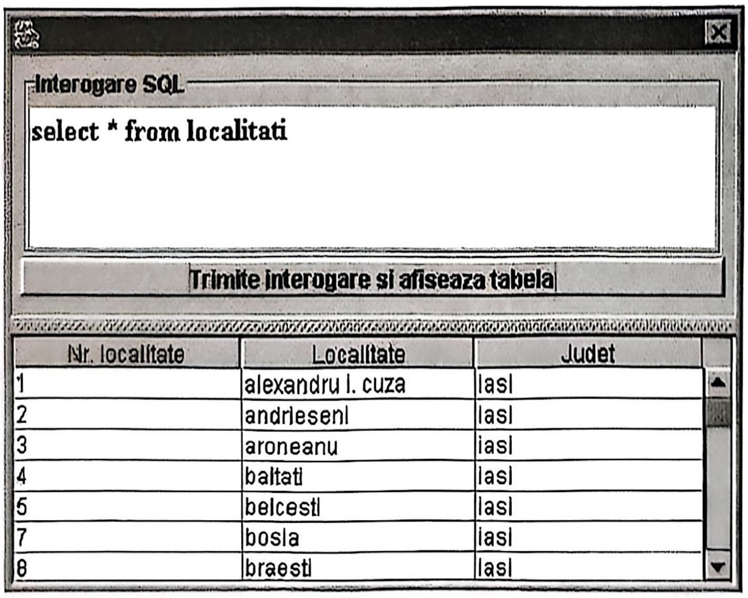
Fereastră pentru interogări SQL

Interfaţa grafică constă din două panouri încadrate într-un `JSplitPane` care le delimitează, unul în care se introduce interogarea într-o zonă de text şi se poate apăsa un buton pentru a se obţine vizualizarea rezultatului, iar celălalt reprezentând panoul în care se va afişa rezultatul, aşa cum se poate vedea în figura următoare. Apăsarea butonului determină crearea modelului tabelei, care este folosit mai apoi pentru a crea un obiect `JTable` afişat într-un `JPanel` (comentariile care însoţesc codul sunt sugestive). Cea mai mare parte a codului este dedicată formatării interfeţei grafice.

```
//interfata cu utilizatorul
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;

public class Interogari extends JDialog {
  JPanel panouJos = new JPanel();
  JPanel panouSus = new JPanel();
```

```
JSplitPane despartitor = new JSplitPane();
JButton butont = new JButton();
JTextArea text;
Dimension marimeEcran;

// constructor care initializeaza prezentarea
public Interogari() {

  super();

  try {
    marimeEcran =
      Toolkit.getDefaultToolkit().getScreenSize();
    setBounds(marimeEcran.width/2200,
      marimeEcran.height/2100, 350,100);
    initDialog();
    pack();
  }
  catch(Exception ex) {      ex.printStackTrace();    }
}

//metoda care face prezentarea tabelei
void initDialog() throws Exception {
  butont.setBorder(BorderFactory.createRaisedBevelBorder());
  butont.setText("Trimite interogare si afiseaza tabela");
  butont.addActionListener( new ActionListener()    {
    public void actionPerformed(ActionEvent e){
      try {
          JDBCAdaptor dt =
          new JDBCAdaptor("starea_civila");
          //creare adaptor
          dt.trimiteInterogare(text.getText());
               //initializarea
           // structurii modelului
          despartitor.add(new JScrollPane(new JTable(dt)),
          JSplitPane.BOTTOM);
          //vizualizare
          }
      catch(Exception exe)
          {   System.err.println("Nu pot realiza operatiile!"
              +exe.toString());   }
    }
              });
  JPanel panouSQL = new JPanel();
  JPanel derulare = new JPanel();
  final JLabel eticheta = new JLabel("SQL:");
```

```
        text = new JTextArea("select * from localitati");
        derulare.setBorder(BorderFactory.createTitledBorder
          (BorderFactory.createEtchedBorder(),"Interogare SQL"));
        derulare.setPreferredSize(new Dimension(400,100));
        text.setFont(new Font("Serif", Font.BOLD, 16));
        text.setLineWrap(true);
        derulare.setLayout(new BorderLayout());
        derulare.add(text);

        panouSQL.setLayout(new BorderLayout());
        panouSQL.add(derulare,BorderLayout.NORTH);
        panouSQL.add(butont,BorderLayout.CENTER);

        panouSus.add(panouSQL, null);
        panouSus.setVisible(true);
        panouJos.setPreferredSize(new Dimension(410, 280));

        despartitor.setOrientation(JSplitPane.VERTICAL_SPLIT);

        despartitor.setTopComponent(panouSus);
        despartitor.add(panouJos, JSplitPane.BOTTOM);
        this.getContentPane().add(despartitor, BorderLayout.CENTER);
    }

    public static void main(String args[])
      {
        Interogari j=new Interogari();
        j.setVisible(true);
      }
}
```



select * from localitati

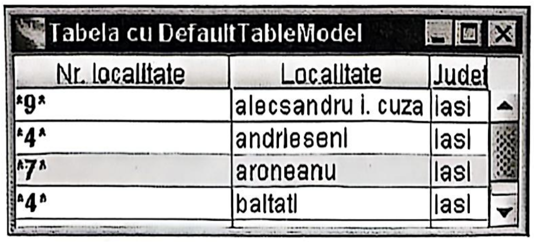
Nr. localitate	Localitate	Judet
1	alexandru i. cuza	iasi
2	andrieseni	iasi
3	aroneanu	iasi
4	baltati	iasi
5	belcesti	iasi
7	bosia	iasi
8	braesti	iasi

Obţinerea rezultatului

În cazul clasei `DefaultTableModel`, accesul spre structura tabelară de date (după cum spuneam, implicit un vector de vectori) se realizează prin intermediul metodelor supraîncărcate de genul `setDataVector()` şi `getDataVector()`. De asemenea, sunt puse la dispoziţia utilizatorilor metode statice de genul `convertToVector()`, care realizează conversii ale tablourilor conţinând instanţe ale clasei `Object` spre tablouri având elementele instanţe ale clasei `Vector`. Drept exemplu de model `DefaultTableModel`, vom adapta primul exemplu descris la începutul capitolului, folosind pentru aceasta unul dintre constructorii clasei `DefaultTableModel`, şi anume cel care are prototipul `public DefaultTableModel(Object[][] liniiDate, Object[] numeColoane)`, ce utilizează intern metoda `setDataVector()` pentru a iniţializa structura tabelară:

```
void creeazaTabela(JFrame f)
{
 Object[][] liniiDate = {
 {"1", "alexandru i. cuza", "iasi"},
  //...
 };
 String[] numeColoane = {"Nr. localitate", "Localitate", "Judet"};
 DefaultTableModel m = new DefaultTableModel(liniiDate,
 numeColoane);
 JTable tabela = new JTable(m);
 JScrollPane jsp = new JScrollPane(tabela);
 f.getContentPane().add(jsp,BorderLayout.CENTER);
 f.setVisible(true);
 }
```

`JTable` permite, în parte, personalizarea vizualizării tabelei fără a fi nevoie de schimbarea *look-and-feel*-ului. Spre exemplu, se pot adăuga pictograme ca fiind conţinut al celulelor, se poate modifica culoarea fundalului celulelor etc. Pentru aceasta trebuie suprascrisă clasa care realizează desenarea tabelei, purtând numele `TableCellRenderer`. Prezentăm un exemplu de personalizare a desenării celulelor pentru una dintre tabelele realizate în acest capitol, prin care conţinutul celulelor unei coloane este desenat folosind culori diferite şi încadrat între două simboluri `"*"`.



Nr. localitate	Localitate	Judet
9	alecsandru i. cuza	Iasi
4	andrieseni	Iasi
7	aroneanu	Iasi
4	baltati	Iasi

Personalizarea tabelelor

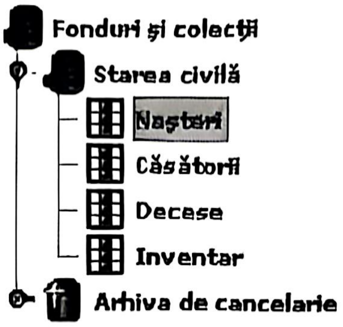
```
class DesenareCelule extends JLabel implements
     TableCellRenderer {
     public Component getTableCellRendererComponent(JTable table,
       // tabela
       Object value, // valoarea celulei
       boolean isSelected, // daca este selectată
       boolean hasFocus, // dacă are focus
       int row, // linia
       int column  // coloana
       )
       {
     if (value!=null)
         {
           int nr;
           String valoare=value.toString();
           try
           {       nr=Integer.parseInt(valoare);
           catch (NumberFormatException e)
           {   nr=1;      }
           if (nr<12)
         setForeground(new Color(10*(12nr),nr*7,13*nr));
         setText("*"+value.toString()+"*");
         setOpaque(isSelected ? true: false);
         }
         return this; // returneaza chiar aceasta eticheta
       }
}
```

Selectarea coloanei care se doreşte personalizată se realizează prin secvenţa:

```
// Personalizare coloana
TableColumnModel tm=tabela.getColumnModel();
for(int i=0;i<tabela.getColumnCount();i++)
{
  TableColumn tc=tm.getColumn(i);
  String s=tabela.getColumnName(i);
  if(s.startsWith("Nr."))
  {
  tc.setCellRenderer(new DesenareCelule());
  }
}
```

13.7.3. `JTree`

Componenta `JTree` permite vizualizarea structurilor arborescente. Clasa `javax.swing.JTree` este clasa de pornire pentru realizarea unui arbore. În afară de aceasta, un întreg pachet stă la baza implementării acestei componente în Swing, şi anume `javax.swing.tree`. Ca şi în cazul `JTable`, o componentă `JTree` nu conţine datele efective, ci doar oferă o vizualizare a datelor ţinute de un model (o clasă care implementează `TreeModel`), eventual cu o formatare dată de un `UIDelegate`. Un exemplu simplu de arbore:



Componenta `JTree`

Metoda care determină această vizualizare este următoarea:

```
public JTree faArbore()
       {
         DefaultMutableTreeNode StareaCivila;
         DefaultMutableTreeNode ArhivaDeCancelarie;
         DefaultMutableTreeNode radacina = new
         DefaultMutableTreeNode("Fonduri \u015Fi colec\u0163ii");
         radacina.add( StareaCivila = new
           DefaultMutableTreeNode("Starea civil\u0103") );
         radacina.add( ArhivaDeCancelarie = new
           DefaultMutableTreeNode("Arhiva de cancelarie") );
         StareaCivila.add( new
           DefaultMutableTreeNode("Na\u015Fteri") );
         StareaCivila.add( new
           DefaultMutableTreeNode("C\u0103s\u0103torii") );
         StareaCivila.add( new
           DefaultMutableTreeNode("Decese") );
         StareaCivila.add( new
           DefaultMutableTreeNode("Inventar") );
         //...
         final JTree arbore = new JTree(radacina);
         DefaultTreeCellRenderer con =
           new DefaultTreeCellRenderer();
         con.setLeafIcon(new ImageIcon("f1.gif"));
         con.setOpenIcon(new ImageIcon("b1.gif"));
         con.setClosedIcon(new ImageIcon("b2.gif"));
         con.setFont(new Font("Verdana",Font.BOLD,10));
         arbore.setCellRenderer(con);
         return arbore;
       }
```

Un `JTree` prezintă nodurile arborelui unul sub altul pe verticală, câte un singur nod pe fiecare rând. Există un singur nod-rădăcină, iar fiecare nod poate avea noduri copii (oricâţi la număr) sau nu. Nodurile care nu au descendenţi poartă numele de noduri-frunză. Nodurile se pot extinde sau restrânge pentru a li se vedea sau nu descendenţii. Operaţiile de extindere/restrângere pot fi tratate prin ataşarea de ascultători arborelui. Fiecare nod prezintă informaţia conţinută într-o celulă a cărei înfăţişare poate fi schimbată folosind o clasă care implementează `TreeCellRenderer`. De asemenea, modul în care utilizatorii vor edita această celulă poate fi controlat şi personalizat prin intermediul unei clase care implementează `TreeCellEditor`.

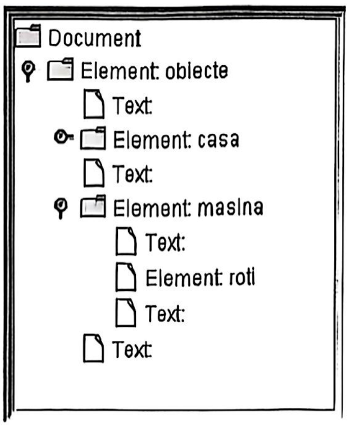
Mecanismul de selecţie a unui nod al unui arbore `JTree` este asemănător cu mecanismul de selecţie din `JList` şi controlat prin intermediul unui model care este o implementare pentru `TreeSelectionModel`.

Interfaţa `TreeModel` stă la baza creării oricărui model de date pentru un `JTree`. Această interfaţă, prin clasele care o implementează, oferă posibilitatea mânuirii structurii de date care dă conţinutul unui `JTree`. Avem la dispoziţie două modalităţi pentru a construi un model de date. Prima ar fi implementarea directă a interfeţei `TreeModel`. Avantajul acestei modalităţi ar fi acela al unei mai mari lejerităţi în alegerea structurilor de date care vor fi alese drept noduri, implicit nodurile fiind instanţe ale clasei `Object`.

A doua ar fi extinderea clasei `DefaultTreeModel` ce reprezintă implementarea implicită pe care Swing o dă interfeţei `TreeModel`. În acest caz, nodurile vor fi instanţe ale unei clase care implementează interfaţa `TreeNode` sau ale unei clase care extinde această interfaţă, şi anume `DefaultMutableTreeNode`. Alegerea uneia sau alteia dintre modalităţi rămâne la latitudinea programatorului.

Vom încerca în continuare să dăm o reprezentare vizuală unui document XML, căruia îi corespunde întotdeauna o structură arborescentă (arbore DOM) ce îi descrie conţinutul. În urma procesării unui document XML folosind DOM, structura arborescentă rezultată va fi o instanţă a clasei `org.w3c.dom.Document`.

Putem spune că tipul de date care implementează `TreeModel` înfăşoară (engl. *wrap*) această structură arborescentă ce se doreşte a fi vizualizată. Pentru aceasta trebuie să suprascriem toate metodele interfeţei `JTreeModel` (şapte la număr) într-o clasă care implementează această interfaţă. Aceste metode arată cum se poate obţine nodul-rădăcină al structurii de date, un anumit nod din structură, numărul de descendenţi ai unui nod, când un nod este frunză sau nu, anunţarea modelului despre o schimbare în structura de date şi adăugarea şi ştergerea de ascultători ai modelului. Pentru amănunte puteţi consulta API-ul DOM şi capitolul dedicat procesării documentelor XML.



Arborele pentru un document XML

```
<?xml version="1.0" encoding="UTF-8"?>
<obiecte>
 <casa>
  <masa>
   <picioare numar="4" />
  </masa>
  <scaun>
  </scaun>
 </casa>
 <masina>
  <roti numar="4" />
 </masina>
</obiecte>
```

Am prezentat documentul XML a cărui vizualizare am realizat-o. Nu am tratat tipurile de elemente care sunt găsite în timpul procesării, pentru a nu complica exemplul.

```
import java.io.*; //pentru intrari/iesiri
import org.w3c.dom.*; //DOM
import javax.xml.parsers.*; //pentru procesorul DOM
import javax.swing.*; //Swing
import javax.swing.tree.*; //pentru JTree
import javax.swing.event.*; //pentru evenimente SWING
import javax.swing.border.*;  //pentru bordere
import java.awt.*; //pentru AWT
import java.awt.event.*; //pentru evenimente AWT


public class PanouArboreDom extends JPanel implements Constants
{
    DocumentBuilderFactory factory=DocumentBuilderFactory.
    newInstance();
    org.w3c.dom.Document document;     //un arbore DOM
    JTree tree;
    JScrollPane treeView;
    InfasuratoareDOM model;

    public PanouArboreDom(String s)
    {
     // cream chenarul
     EmptyBorder eb = new EmptyBorder(1,1,1,1);
     BevelBorder bb = new BevelBorder(BevelBorder.LOWERED);
     CompoundBorder cb = new CompoundBorder(eb,bb);
     this.setBorder(new CompoundBorder(cb,eb));
     //cream un arbore DOM din document
     try
```

```
      {
         DocumentBuilder builder=factory.newDocumentBuilder();
         document=builder.parse(new File(s));
         document.getDocumentElement().normalize();
         model = new InfasuratoareDOM();
         tree = new JTree(model);
         tree.getSelectionModel().setSelectionMode(
           TreeSelectionModel.SINGLE_TREE_SELECTION);
         //cream panoul de derulare
         treeView = new JScrollPane(tree);

         this.setLayout(new BorderLayout());
         this.add("Center",treeView );
      }
      catch (Exception pce)
      {
        System.err.println("Eroare"+pce);
        }
    } // constructor


    //infasoara structura arborescenta rezultata in urma
    //parsarii documentului XML
    public class InfasuratoareDOM implements TreeModel
    {
      //se da implementare metodelor din interfata TreeModel

      //returneaza radacina
      public Object  getRoot() {
        return new InfasuratoareNOD(document);
      }

      //returneaza true daca nodul dat este frunza
      public boolean isLeaf(Object unNod) {
        InfasuratoareNOD nod = (InfasuratoareNOD) unNod;
        if (nod.numarCopii() > 0) return false;
        return true;
      }

      //returneaza numarul de copii pentru un nod dat
      public int getChildCount(Object parinte) {
        InfasuratoareNOD nod = (InfasuratoareNOD) parinte;
        return nod.numarCopii();
      }

      //returneaza copilul cu indicele nr
```

```
  public Object  getChild(Object parinte, int nr) {
    InfasuratoareNOD nod = (InfasuratoareNOD) parinte;
    return nod.copilul(nr);
  }

  //returneaza indicele copilului copil pentru parintele
  //parinte
  public int getIndexOfChild(Object parinte, Object copil) {
    InfasuratoareNOD nod = (InfasuratoareNOD) parinte;
    return nod.index((InfasuratoareNOD) copil);
  }

  public void valueForPathChanged(TreePath path, Object
  newValue) {
   //nu scriem nimic. daca nu face modificari in arbore pe
   //calea data de path
   System.out.println("modificat");
   InfasuratoareNOD  aNode = (InfasuratoareNOD)path.
     getLastPathComponent();
    org.w3c.dom.Node sampleData = aNode.NodDOM;
    sampleData.setNodeValue((String)newValue);
  }

  //pentru a adauga ascultatori personalizati
  public void addTreeModelListener( TreeModelListener
  listener ) { }

  //pentru a sterge ascultatori personalizati
  public void removeTreeModelListener( TreeModelListener
  listener ) { }
 }

//aceasta clasa infasoara (wrap) un nod din org.w3c.dom.Node,
//dand posibilitatea suprrascrierii metodei toString
class InfasuratoareNOD implements Constants
{
  boolean compress=true;
  int ind;
  //contine o variabila membru de tip Node si metode pentru
  //lucrul cu ea
  org.w3c.dom.Node NodDOM;

  //contructor care duce un nod DOM intr-un nod
  //InfasuratoareNOD
  public InfasuratoareNOD(org.w3c.dom.Node nod) {
```

```
  NodDOM = nod;
}

//suprascriem toString din Object pentru a personaliza
//ceea ce va aparea in celulele vizualizarii JTree
public String toString() {
  String s = typeName[NodDOM.getNodeType()];
  String numeNOD = NodDOM.getNodeName();
  if (! numeNOD.startsWith("#")) {
     s += ": " + numeNOD;
  }

  if (NodDOM.getNodeValue() != null) {
     if (s.startsWith("ProcInstr"))
        s += ", ";
     else
        s += ": ";

     String t = NodDOM.getNodeValue().trim();
     int x = t.indexOf("\n");
     if (x >= 0) t = t.substring(0, x);
     s += t;
  }
  return s;
}

//in continuare se definesc o serie de metode
//care faciliteaza accesul la modurile arborelui

// returneaza indexul unui copil
public int index(InfasuratoareNOD copilul) {
  int numar = numarCopii();
  for (int i=0; i<numar; i++) {
    InfasuratoareNOD n = this.copilul(i);
    if (copilul == n) return i;
  }
  return 1; // in caz contrar
}

//da numarul de copii pentru un nod DOM
public int numarCopii()
{
    return NodDOM.getChildNodes().getLength();
}

//returneaza copilul cu un anumit index pentru un nod DOM
```

```
      public InfasuratoareNOD copilul(int indexDeCautare) {
          org.w3c.dom.Node nod =
            NodDOM.getChildNodes().item(indexDeCautare);
          return new InfasuratoareNOD(nod);
      }
    }
}
```

De observat că `TreeModel` ar putea fi folosită ca o manieră de a crea şi stoca structuri arborescente, fără a le da o reprezentare vizuală.

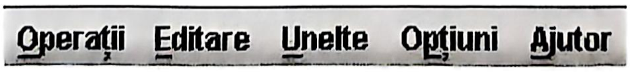
13.8. Meniuri şi bare de unelte

13.8.1. Meniuri

Meniurile reprezintă modalitatea standard folosită în interfeţele grafice pentru a permite utilizatorului aplicaţiei alegerea unei opţiuni care generează o acţiune, din mai multe posibile. Într-o aplicaţie, meniurile apar fie grupate într-o bară de meniu, fie ca meniuri *pop-up*, care se vizualizează doar în momentul apăsării unui buton al mouse-ului.

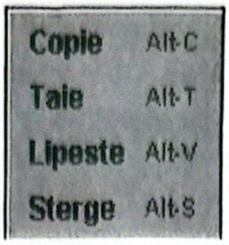
Un meniu poate avea ca opţiuni alte submeniuri. Este indicat să nu se depăşească trei niveluri în adâncime şi, de asemenea, ca un meniu să nu conţină mai mult de şapte opţiuni pentru o bună funcţionare a aplicaţiei.

`JMenuBar` reprezintă un container pentru mai multe elemente `JMenu` aşezate într-o bară orizontală în partea superioară a unei ferestre. Pentru a adăuga elemente `JMenu` unui meniu bară folosim metode cum ar fi `add(JMenu meniu)`. `JMenuBar` foloseşte `DefaultSingleSelectionModel` drept model pentru selecţie, ceea ce determină ca un singur element `JMenu` să fie selectat la un moment dat. Deoarece `JMenuBar` moşteneşte `JComponent`, poate fi plasat oriunde în interiorul unui container. Pentru a adăuga un meniu bară unei aplicaţii se foloseşte metoda `setJMenuBar()`, aplicabilă tuturor containerelor care implementează `RootPaneContainer` (a se vedea subcapitolul dedicat containerelor de bază).



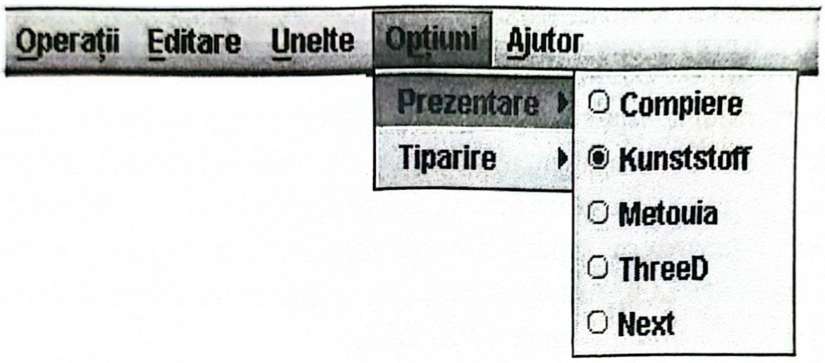
Meniu de tip bară

`JPopupMenu` reprezintă o mică fereastră, prezentând o coloană de elemente – opţiuni din care se aşteaptă ca utilizatorul să aleagă una. Adăugarea sau ştergerea de elemente se realizează folosind metoda `add()`, respectiv `remove()`. Folosind metoda `show()`, putem determina ca un meniu *pop-up* să devină vizibil la coordonatele specificate ale unei componente, după ce, în prealabil, a fost creat. Această componentă se numeşte apelant (engl. *invoker*) şi se poate schimba folosind un apel `setInvoker (Component apelant)`. `JPopupMenu` detectează dacă componenta apelantă este un container de bază sau nu, comportându-se în consecinţă. Un exemplu complet de utilizare a meniurilor *pop-up* aveţi în secţiunea dedicată componentelor text.



Meniu *pop-up*

`JMenuItem` reprezintă o opţiune dintr-un meniu şi este elementul de bază din compoziţia meniurilor. Acestui element, pe lângă text, i se pot adăuga drept conţinut pictograme. De asemenea, fiecărei opţiuni (un element `JMenuItem`) i se poate asocia o tastă (engl. *mnemonic*) folosind metoda `setMnemonic()` moştenită din clasa `AbstractButton`. Dacă litera care dă tasta asociată se regăseşte în interiorul unei opţiuni, atunci va fi subliniată prima apariţie a ei. Apăsarea tastei *Alt* (depinde de *look-and-feel*) împreună cu aceasta determină activarea opţiunii.



Exemple de meniuri

De asemenea, unei opţiuni meniu îi putem ataşa un accelerator care va apărea ca text micşorat în partea dreaptă a elementului `JMenuItem`. Diferenţa dintre *mnemonic* şi accelerator este aceea că un accelerator poate fi folosit chiar dacă meniul care conţine opţiunea nu este vizibil la momentul curent, singura condiţie necesară fiind ca fereastra ce conţine meniul să fie activă. Ataşarea acceleratorului se realizează folosind un apel de tipul: `element.setAccelerator( KeyStroke.getKeyStroke (KeyEvent.VK_A,  KeyEvent.CTRL_MASK,  false)`, ceea ce va determina ca opţiunea să se activeze prin apăsarea combinaţiei de taste *Ctrl+A*.

Pentru a grupa opţiunile ce se doresc vizualizate într-o bară de meniu, se foloseşte clasa `JMenu` care, surprinzător, extinde `JMenuItem`. Putem privi `JMenu` ca pe un `JMenuItem` care determină apariţia unui meniu *pop-up* prin selectare. Instanţe ale clasei `JMenu` pot fi adăugate unui alt `JMenu` pentru a crea un submeniu al acestuia sau al meniului bară. Primul caz este ilustrat în figura precedentă, operaţia de adăugare determinând apariţia unei săgeţi ce semnifică faptul că această opţiune are un submeniu. Al doilea caz este ilustrat în figura care urmează. Prin alegerea opţiunii `Editare` din bara de meniu se va desfăşura un meniu *pop-up* conţinând trei opţiuni. Acceleratorii pentru `JMenu` sunt dezactivaţi chiar dacă extinde clasa `JMenuItem`. Clasa `JMenu` conţine un meniu *pop-up*, cel ce se va afişa în momentul selectării cu mouse-ul a numelui meniului. Acest meniu *pop-up* îl putem obţine folosind metoda `getPopupMenu()`. Îi putem schimba locul afişării utilizând metoda `setMenuLocation()`.

`JMenu` permite adăugarea de elemente `JMenuItem`, instanţe ale claselor implementând interfaţa `Action` (despre care vom discuta la finalul acestui subcapitol), alte componente sau şiruri de caractere, folosind supraîncărcările metodei `add()`. De asemenea, între opţiunile unui meniu se pot adăuga linii despărţitoare recurgând la metoda `addSeparator()`, pentru a evidenţia, spre exemplu, un grup de opţiuni. Apelul metodei `addSeparator()` determină crearea unei componente `JSeparator`, reprezentând o zonă goală care este adăugată sub elementul curent al meniului. Posibilitatea de a adăuga alte componente drept opţiuni ale unui meniu (apelul de adăugare are forma `add(Component comp)`) poate fi folositoare, de exemplu, dacă dorim să adăugăm o componentă ceas sau oricare componentă personalizată.

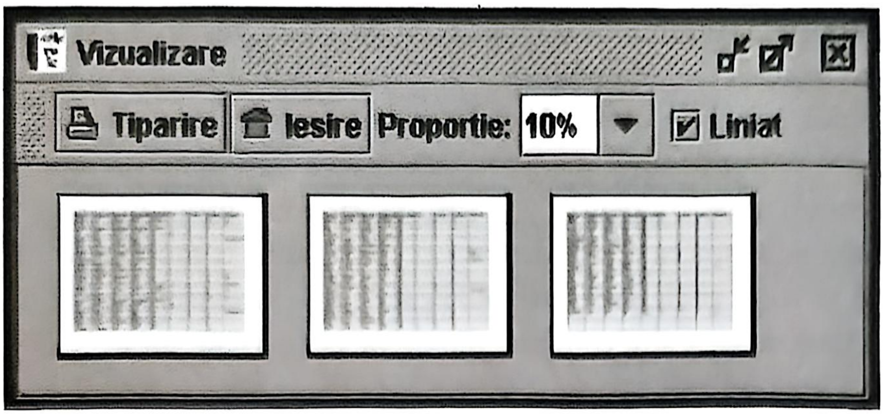
Clasa `JRadioButtonMenuItem` este folosită pentru adăugarea de butoane radio drept opţiuni în meniuri. Butoanele pot fi selectate utilizând metoda `setSelected()` şi li se pot ataşa ascultători de tipul `ActionListener` sau `ChangeListener`. Pentru a permite setarea unui singur buton la un moment dat, trebuie să grupăm butoanele folosind clasa `ButtonGroup`. Un exemplu complet referitor la acest subiect găsiţi în secţiunea dedicată *look-and-feel*-ului. De asemenea, pentru a putea adăuga căsuţe selectabile putem utiliza clasa `JCheckBoxMenuItem`.

Pentru a intercepta setarea de către utilizator a unei opţiuni dintr-un meniu, de obicei, ataşăm acesteia un ascultător `ActionListener`. O alternativă ar fi folosirea instanţelor clasei `Action`, care permit crearea de elemente opţiuni, înglobând şi modul de tratare a selectării lor. Această modalitate este folosită, de exemplu, în cazul în care o aceeaşi acţiune se regăseşte în acelaşi timp şi într-un meniu, şi într-o bară de unelte, permiţând implementarea de cod pentru tratarea acesteia o singură dată.

13.8.2. `JToolBar`

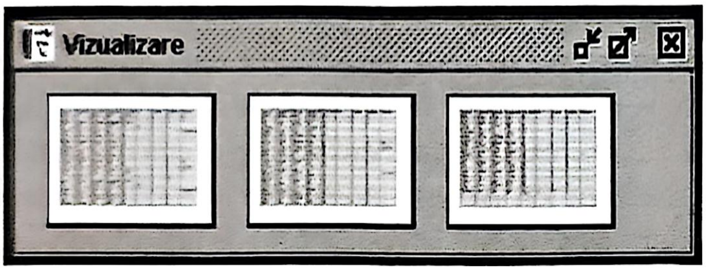
Bara de unelte (engl. *toolbar*) reprezintă o colecţie de pictograme grupate într-o bară situată, în general, sub bara de meniu. Aceste pictograme sunt, de obicei, acceleratori pentru anumite opţiuni din meniul de bază al aplicaţiei, deoarece este mai uşor să apăsăm un *icon* aflat deja pe suprafaţa de lucru decât să derulăm meniul. Bara de unelte poate fi folosită, aşa cum îi spune şi numele, şi ca un mijloc de a oferi utilizatorilor diverse unelte cu care aceştia să lucreze pe parcursul rulării aplicaţiei. Spre exemplu, în aplicaţiile grafice, sunt puse la dispoziţia utilizatorilor diverse instrumente pentru desenare, alegerea pictogramelor cu un design corespunzător determinând schimbarea cursorului mouse-ului şi a funcţionalităţii. O ultimă utilitate a barei de unelte ar fi pentru navigare. Un bun exemplu pentru acest caz ar putea fi un browser Web, trecerea de la o pagină la alta realizându-se prin pictogramele din bara de navigare.

Pentru implementarea barei de unelte în Swing avem la dispoziţie clasa `JToolBar`. Unei bare de unelte îi putem adăuga, folosind metoda `add()`, orice componentă, aşa cum se poate vedea în figura următoare, ilustrând o fereastră de vizualizare înaintea imprimării, în care avem două butoane `JButton`, un `JComboBox`, un buton `JCheckBox`.

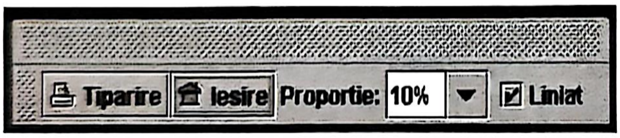


Utilizarea barelor de unelte

De asemenea, folosind mouse-ul putem poziţiona bara de unelte oriunde în interiorul ferestrei care o conţine şi chiar să o separăm de fereastră, unde o putem reintroduce folosind mouse-ul. Atunci când se află în exteriorul unei ferestre, bara de unelte este plasată într-o fereastră `JFrame`. Cele prezentate se pot observa în imaginile care urmează.



Bara pentru vizualizare

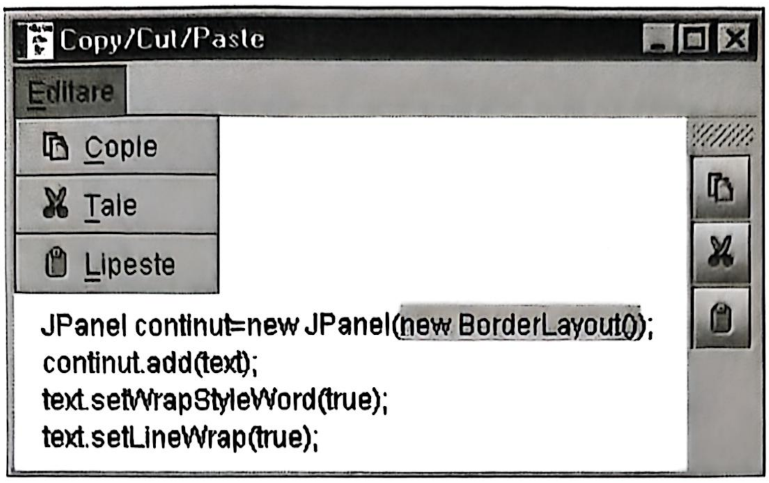


Bara pentru tipărire

Între elementele unei bare de unelte se pot adăuga separatori folosind metoda `addSeparator()`. Drept exemplu de utilizare a barelor de unelte aveţi la dispoziţie aplicaţia de la subcapitolul ce descrie desenarea componentelor grafice.

Vom discuta, în continuare, despre interfaţa `Action`, care oferă o modalitate comună atât meniurilor, cât şi barelor de unelte pentru a adăuga acţiuni reprezentate de butoane `JButton`, în cazul `JToolBar`, sau opţiuni `JMenuItem`, în cazul `JMenu`, `JPopupMenu`, cu diverse funcţionalităţi, folosind metoda `add(Action actiune)`. Interfaţa `Action` extinde interfaţa `ActionListener`, deci pune la dispoziţia programatorilor metoda `actionPerformed()`, prin a cărei suprascriere se pot trata evenimentele `ActionEvent` generate de activarea acţiunilor. În practică este folosită clasa `AbstractAction` din `javax.swing`, care este o implementare abstractă a interfeţei `Action`. Constructorii clasei `Action` permit vizualizarea de text sau pictograme ca reprezentare vizuală a acţiunii.

În aplicaţia care urmează vom recurge la toate noţiunile acestui subcapitol:



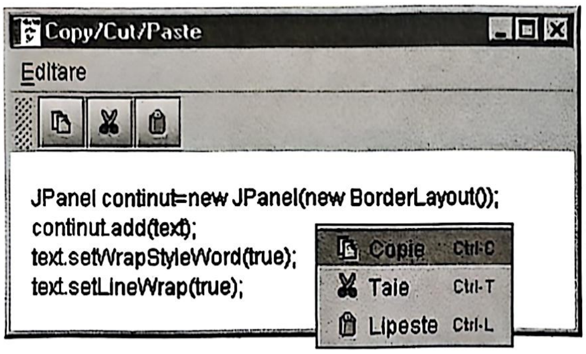
```
JPanel continut=new JPanel(new BorderLayout());
continut.add(text);
text.setWrapStyleWord(true);
text.setLineWrap(true);
```

Aplicaţie cu bară de unelte

```
/* Fereastra aplicatiei */
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.border.*;
public class FereastraAplicatie extends JFrame {
   //variabile de clasa
   static JMenuBar meniuBara;
   final JTextArea text=new JTextArea();
   Image img;
   int pozitie=230;
   //constructorul
   public FereastraAplicatie() throws Exception {
       super("Copy/Cut/Paste");
       // pozitionam fereastra si-i setam dimensiunile
       Toolkit t=this.getToolkit();
       Dimension marimeEcran = Toolkit.getDefaultToolkit().
           getScreenSize();
       setBounds ( pozitie, pozitie, marimeEcran.width 2*
           pozitie, marimeEcran.height/230);
       // setam pictograma din coltul din stanga-sus al ferestrei
       img=t.getImage("icon.jpg");
       this.setIconImage(img);
       this.getContentPane().setLayout(new BorderLayout());
       // construim partile componente
       doMeniu();
       doBaraUnelte();
       doContinut();
}

   // construim continut
```

```
protected void doContinut() throws Exception
{
  JPanel continut=new JPanel(new BorderLayout());
  continut.add(text);
  text.setWrapStyleWord(true);
  text.setLineWrap(true);

  text.addMouseListener(new MouseAdapter()
     {
     public void mousePressed(MouseEvent e)
     {
        int i=e.getClickCount();
        if (e.getButton()==3 && i==1)
        {
          JPopupMenu pop=doMeniuPopup();
          pop.show(text, e.getX(), e.getY());
          pop.setVisible(true);
     }}});
  this.getContentPane().add(continut);  // in acest panou vom
  //adauga continutul ferestrei
  }

protected void doMeniu() {
  meniuBara = new JMenuBar(); // cream bara de meniu
  meniuBara.setOpaque(true);
  JMenu editare = doMeniuEditare(); //cream un meniu
                                    //optiuni
  editare.setMnemonic('E'); // definim shortcut (Alt+E)
  meniuBara.add(editare);     // adaugam meniul optiuni la
                              // bara de meniu
  setJMenuBar(meniuBara);
  }

// construim meniul editare
protected JMenu doMeniuEditare() {
  JMenu editeaza = new JMenu("Editare");
  JMenuItem Icopie=editeaza.add(new Copie("Copie",new
     ImageIcon("copie.gif")));
  editeaza.addSeparator();
  JMenuItem Itaie=editeaza.add(new Taie("Taie",new
     ImageIcon("taie.gif")));
  editeaza.addSeparator();
  JMenuItem Ilipeste=editeaza.add(new Lipeste("Lipeste",
     new ImageIcon("lipeste.gif")));
  Icopie.setMnemonic('C');
  Itaie.setMnemonic('T');
  Ilipeste.setMnemonic('L');
  return editeaza;
}

JPopupMenu doMeniuPopup()
{
  final JPopupMenu editeaza=new JPopupMenu("Editare");
  JMenuItem Icopie=editeaza.add(new Copie("Copie",new
     ImageIcon("copie.gif")));
  JMenuItem Itaie=editeaza.add(new Taie("Taie",new
     ImageIcon("taie.gif")));
  JMenuItem Ilipeste=editeaza.add(new Lipeste("Lipeste",
     new ImageIcon("lipeste.gif")));
  Icopie.setAccelerator(KeyStroke.getKeyStroke (KeyEvent.
     VK_C,ActionEvent.CTRL_MASK));
  Itaie.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_T,
     ActionEvent.CTRL_MASK));
  Ilipeste.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.
     VK_L, ActionEvent.CTRL_MASK));
  return editeaza;
}

void doBaraUnelte()
{
  JToolBar tb = new JToolBar();
  tb.add(new Copie("",new ImageIcon("copie.gif")));
  tb.add(new Taie("",new ImageIcon("taie.gif")));
  tb.add(new Lipeste("", new ImageIcon("lipeste.gif")));
  this.getContentPane().add(tb,BorderLayout.NORTH);
}


// actiuni definite
class Copie extends AbstractAction
{
  Copie(String nume, ImageIcon icon)
     { super(nume, icon); }
  public void actionPerformed(ActionEvent e)
     { text.copy();   }
}

class Taie extends AbstractAction
{
  Taie(String nume, ImageIcon icon)
```

```
      { super(nume, icon); }
  public void actionPerformed(ActionEvent e)
      { text.cut();    }
}

class  Lipeste extends AbstractAction
{
  Lipeste(String nume, ImageIcon icon)
      { super(nume, icon); }
  public void actionPerformed(ActionEvent e)
      { text.paste();    }
}
public void iesire() {
  System.exit(0); // parasim aplicatia
  }
}
```



```
JPanel continut=new JPanel(new BorderLayout());
continut.add(text);
text.setWrapStyleWord(true);
text.setLineWrap(true);
```

Mutarea barei de unelte

13.9. Dialoguri

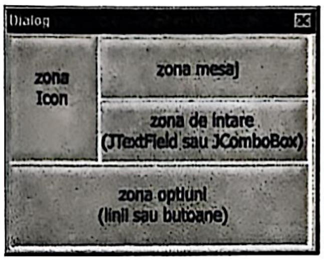
Dialogurile sunt necesare în cazul în care se doreşte avertizarea utilizatorului asupra unui eveniment care s-a produs în aplicaţie sau în cazul în care se doreşte validarea unor date introduse de utilizator. În acest ultim caz, fereastra de dialog va avea un buton de acceptare (engl. *accept*) care va putea fi apăsat doar dacă datele introduse de utilizator sunt valide. O altă utilizare a dialogurilor ar fi pentru a selecta dintr-o listă un set de elemente care vor fi adăugate în altă listă. Spre exemplu, prima listă poate reprezenta o mulţime de programatori dintr-o firmă, iar a doua listă poate fi formată doar din aceia care lucrează la un anumit proiect. Sunt şi alte utilizări ale ferestrelor de dialog, dar este indicată folosirea lor cu grijă pentru a nu suprasolicita utilizatorul. Pentru a ocoli ferestrele de dialog care să prezinte utilizatorilor mesaje, acestea se pot afişa într-o bară de stare a ferestrei principale.

În una dintre secţiunile precedente am prezentat componenta `JDialog`, care reprezintă metoda standard de a crea ferestre de dialog şi care permite realizarea de dialoguri complexe. Mai avem la dispoziţie şi alte componente, cum ar fi `JOptionPane`, pentru a realiza dialoguri simple şi într-o formă standard. De asemenea, în Swing sunt implementate şi câteva dialoguri standard personalizabile, cum ar fi cel pentru alegerea (engl. *choose*) unui fişier, şi anume `JFileChooser`, respectiv pentru selectarea unei culori `ColorChooser`. Aceste componente le vom prezenta în continuare.

13.9.1. `JOptionPane`

Această clasă reprezintă o metodă facilă de a realiza dialoguri utile pentru a prezenta un mesaj, pentru a prezenta o întrebare utilizatorului şi a aştepta introducerea răspunsului. Trebuie ştiut că fiecare dialog `JOptionPane` este modal (excepţie fac dialogurile din ferestrele interne `JInternalFrame`), ceea ce înseamnă că firul de execuţie al aplicaţiei va fi blocat atât timp cât fereastra de dialog este vizualizată. De asemenea, `JOptionPane` extinde direct clasa `JComponent` şi nu `JDialog` aşa cum ne-am aştepta, având rol de container intermediar, după cum îi spune şi numele, care poate fi aşezat într-o fereastră `JFrame`, `JDialog` sau într-o fereastră interioară `JInternal Frame`.

Forma standard pe care o are un dialog `JOptionPane` este următoarea:



Structura ferestrei `JOptionPane`

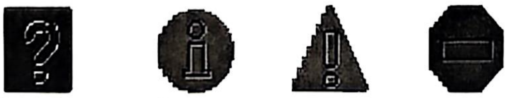
Se pot crea dialoguri folosind constructorii clasei `JOptionPane`, care sunt adăugate în `JDialoguri` făcute mai apoi vizibile. De asemenea, se pot crea dialoguri recurgând la metode statice de forma `showXXXDialog()` ale clasei `JOptionPane`. Există patru tipuri de dialoguri, fiecare având metode specifice de afişare. Urmează prezentarea lor împreună cu metodele statice prin care se realizează activarea lor, respectiv explicaţiile corespunzătoare.

Tip mesaj	Metoda	Descriere
Mesaj	**`showConfirmDialog`**`()`	Această metodă afişează un dialog cu câteva butoane şi returnând un `int`, reprezentând butonul apăsat. Metoda are patru supraîncărcări, permiţând introducerea unei componente-părinte, a unui mesaj, titlu, tip opţional, a tipului mesajului, a unei pictograme.
Confirmare	**`showInputDialog`**`()`	Această metodă afişează un dialog folosit pentru a obţine date de la utilizator şi returnează un `String`, când componenta de intrare este un câmp text sau un `Object`, când componenta de introducere este o listă sau un *combobox*. Sunt şase supraîncărcări ale metodei cerând introducerea unei componente-părinte, a unui mesaj, titlu, a tipului opţiunii, tipului mesajului, a unei pictograme, tablou de selecţii posibile şi a unui element selectat iniţial. Mai sunt prezentate două butoane în zona opţională, şi anume `OK` şi `Cancel`.

➔

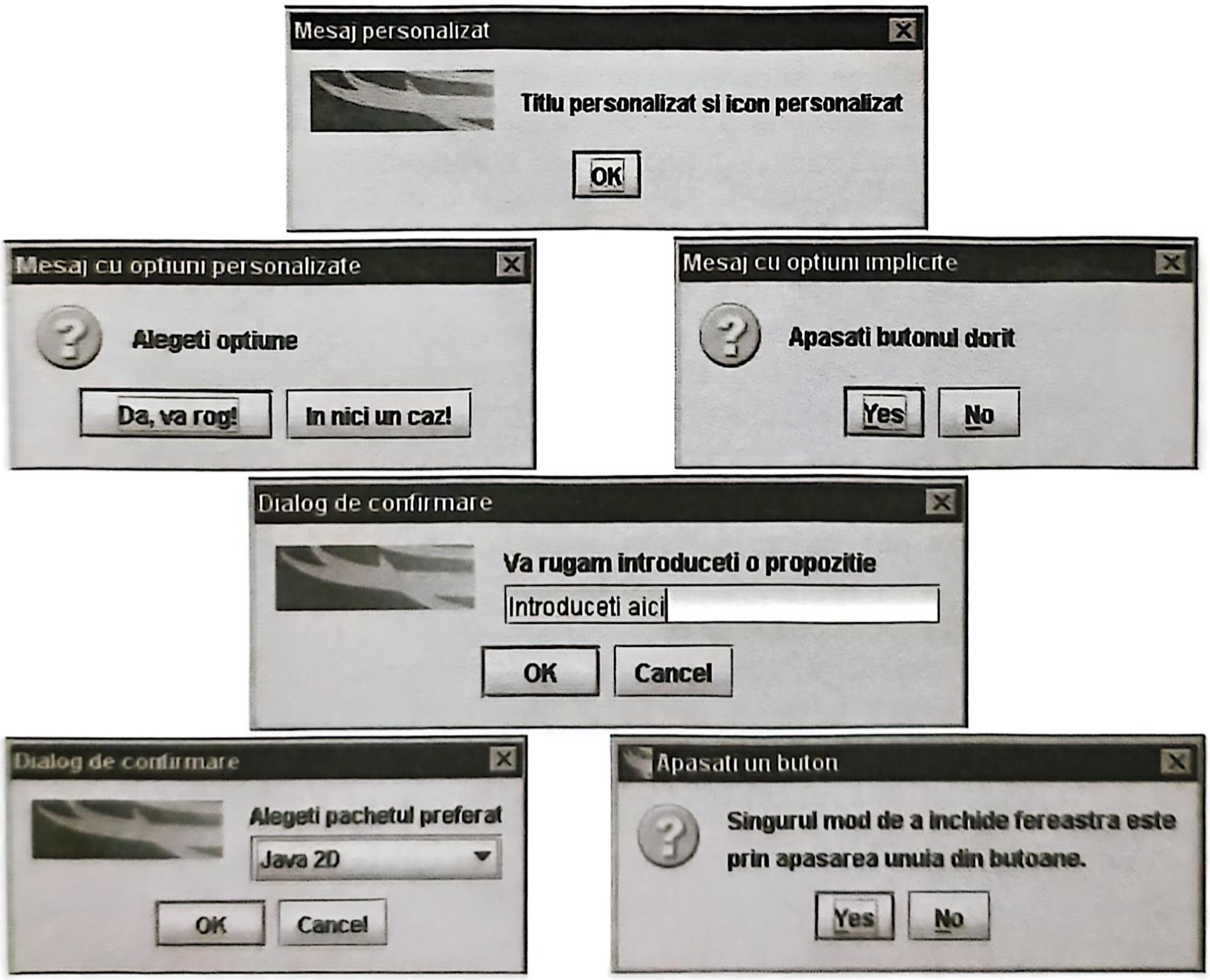
Intrare	**`showMessageDialog()`**	Această metodă afişează un dialog cu un buton `OK` şi nu returnează nimic. Sunt trei supraîncărcări ale metodei cerând introducerea unei componente-părinte, a unui mesaj, titlu, tip de mesaj, pictogramă.
Opţiune	**`showOptionDialog()`**	Această metodă afişează un dialog care poate fi personalizat mai mult decât celelalte dialoguri, returnând un index dintr-un tablou de opţiuni `Object` specificate. Există doar o metodă care primeşte o componentă-părinte, un tip de opţiune, un tip de mesaj, o pictogramă, un tablou de opţiuni `Object` şi un obiect `Object` reprezentând focusul iniţial.

Prezentăm în continuare explicaţiile parametrilor care pot apărea în apelul metodelor de tipul `showXXXDialog()`. Pentru amănunte despre toate supraîncărcările acestor metode puteţi să consultaţi *javadoc*-ul.

Parametri	Explicaţie
o componentă--părinte	Reprezintă componenta care generează dialogul. Dacă părintele este `JFrame`, atunci dialogul va fi plasat într-un `JDialog` care va fi centrat relativ la fereastra-părinte. Dacă părintele este `null`, atunci dialogul se va centra relativ la ecran.
un mesaj `Object`	Reprezintă un mesaj care va fi afişat în zona pentru mesaj. În general, este vorba de un `String`, aşezat pe linii diferite folosind caracterul `'/n'`. Dacă mesajul nu este `String`, ci un alt obiect, acest lucru se tratează în felul următor : – `Icon` va fi vizualizat într-o etichetă `JLabel`. – `Component` va fi plasată în zona pentru mesaj. – `Object[]` vor fi plasate vertical într-o coloană verticală (rezultatul apelării `toString()`). – `Object` – se va afişa textul rezultat ca la apelul `toString()`.
un întreg care dă tipul mesajului	Poate avea una dintre valorile: `ERROR_MESSAGE`, `INFORMATION MESSAGE`, `WARNING_MESSAGE`, `QUESTION_MESSAGE`, `PLAIN_MESSAGE`. Această constantă spune *look-and-feel*-ului ce *icon* va fi afişat în zona destinată *icon*-ului. *Icon*-urile implicite pentru tipurile de mesaje enumerate sunt:  întrebare, informaţii, avertisment, eroare
un întreg care dă tipul opţiunii	Poate avea una dintre valorile: `DEFAULT_OPTION`, `YES_NO_OPTION`, `YES_NO_CANCEL_OPTION`, `OK_CANCEL_OPTION`. Această constantă spune ce butoane vor fi afişate în zona pentru opţiuni. Un alt set de opţiuni este reprezentat de ceea ce pot returna metodele `showXXXDialog()` prezentate în tabelul precedent. Acestea sunt `CANCEL_OPTION`, `CLOSED_OPTION`, `NO_OPTION`, `OK_OPTION`, `YES_OPTION`. De remarcat, `CLOSED_OPTION` este returnat când butonul de închidere de pe bara de text este apăsat.

un element `Icon`	Este pictograma afişată în zona pentru *icon*. Dacă nu este specificat pictograma ce se va afişa, va fi determinată de *look-and-feel*.
un tablou de obiecte `Object`	Putem specifica un tablou de obiecte care vor fi afişate în zona pentru opţiuni. Aceste obiecte pot fi şiruri `String` sau instanţe `Icon` care vor fi folosite pentru a fi decorate butoanele `JButton`, dar şi instanţele `Component`, care vor fi afişate toate pe un rând în zona pentru opţiuni. De asemenea, se va folosi `toString()` pentru a determina şirul care se va afişa.
o valoare iniţială `Object`	Va preciza care componentă din zona pentru opţiuni va deţine focusul.
un tablou de valori `Object` selectabile	Specifică un tablou de opţiuni din care se poate selecta una. Dacă tabloul are mai puţin de 12 opţiuni, se va folosi `JList` pentru prezentare, altfel se va utiliza `JComboBox`.
un titlu `String`	Reprezintă titlul care se va afişa pe bara de titlu.

Prezentăm o aplicaţie demonstrativă care activează mai multe tipuri de dialoguri `JOptionPane` realizate prin metode de tipul `showXXXDialog()`. Ultimul dialog este realizat folosind constructorul pentru `JOptionPane`.



Mesaj personalizat

Titlu personalizat si icon personalizat

Mesaj cu optiuni personalizate

Alegeti optiune

Mesaj cu optiuni implicite

Apasati butonul dorit

Dialog de confirmare

Va rugam introduceti o propozitie

Introduceti aici

Dialog de confirmare

Alegeti pachetul preferat

Java 2D

Apasati un buton

Singurul mod de a inchide fereastra este prin apasarea unuia din butoane.

Exemple de ferestre `JOptionPane`

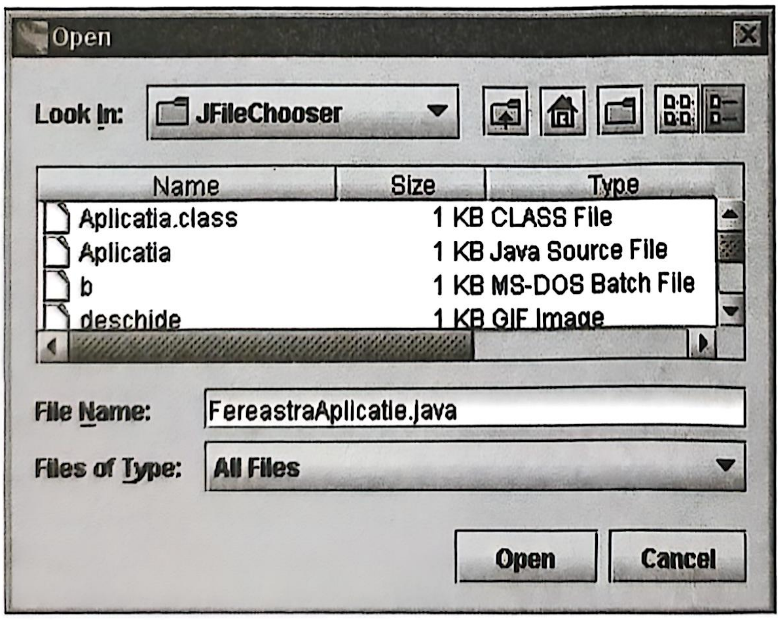
```
/*Aplicatia*/

import javax.swing.*;
import java.beans.*;

public class Aplicatia {

  public static void main( String[] args )
  {
    try
    {
     UIManager.setlook-and-feel(
      "com.incors.plaf.kunststoff Kunststoff look-and-feel"
     FereastraAplicatie fereastra =
      new FereastraAplicatie();
     fereastra.setVisible(true);
     {
      Icon icon=new ImageIcon("icon.jpg");

      // dialog mesaj
      JOptionPane.showMessageDialog(fereastra,
       "Titlu personalizat si icon personalizat", "Mesaj
       personalizat", JOptionPane.INFORMATION_MESSAGE,
       icon);

      // dialog optiuni
      Object[] optiuni = {"Da, va rog!", "In nici un caz!"};
      int nr = JOptionPane.showOptionDialog(fereastra,
       "Alegeti optiune", "Mesaj cu optiuni personalizate",
       JOptionPane.YES_NO_OPTION, JOptionPane.QUESTION_
       MESSAGE, null, optiuni, optiuni[0]);
      System.out.println("S-a ales butonul: "+optiuni[nr]);

      // dialog confirmare
      int numarul = JOptionPane.showConfirmDialog(fereastra,
       "Apasati butonul dorit", "Mesaj cu optiuni implicite",
       JOptionPane.YES_NO_OPTION);
      System.out.println("S-a ales butonul: "+optiuni
       [numarul]);

      //dialog de intrare
      String iesire = (String)JOptionPane.showInputDialog
       (fereastra,"Va rugam introduceti o propozitie", "Dialog
       de confirmare", JOptionPane.INFORMATION_MESSAGE, new
```

```
 ImageIcon("icon.jpg"), null, "Introduceti aici");
System.out.println("S-a introdus: "+iesire);

// dialog de intrare folosind ComboBox
String[] pachete = new String[] {"AWT", "Swing",
 "Accessibility", "Java 2D", "Drag and Drop" };
String iesirea = (String)JOptionPane.showInputDialog(
 fereastra, "Alegeti pachetul preferat",
 "Dialog de confirmare", JOptionPane.INFORMATION_MESSAGE,
 new ImageIcon("icon.jpg"), pachete, "Java 2D");
System.out.println("S-a introdus: "+iesirea);

// dialog realizat prin constructor
final JOptionPane panouOptiuni = new JOptionPane(
 "Singurul mod de a inchide fereastra este\n"+
 "prin apasarea unuia din butoanele.\n" +
 "Mesaj cu obtiuni", JOptionPane.QUESTION_MESSAGE,
 JOptionPane.YES_NO_OPTION);
final JDialog dialog = new JDialog(fereastra,
 "Apasati un buton", true);
dialog.setContentPane(panouOptiuni);
panouOptiuni.addPropertyChangeListener(
 new PropertyChangeListener() {
  public void propertyChange(PropertyChangeEvent e)
{
    String prop = e.getPropertyName();
    if (dialog.isVisible() && (e.getSource() == panouOptiuni)
      && (prop.equals(JOptionPane.VALUE_PROPERTY) ||
      prop.equals(JOptionPane.INPUT_VALUE_PROPERTY)))
      {
       dialog.setVisible(false);
      }
  }
});

dialog.pack();
dialog.setVisible(true);

int valoare = ((Integer)panouOptiuni.getValue()).
 intValue();
if (valoare == JOptionPane.YES_OPTION)
{
    System.out.println("S-a ales butonul <Da> ");
}
else
```

```
      System.out.println("S-a ales butonul <Nu> ");
      }

    }
    catch(Exception ex)
    {
     System.err.println("Eroare"+ex.toString());
    }
  }
}
```

13.9.2. **JFileChooser**

Swing introduce un mod standard pentru navigarea în structura de directoare şi selecţia de fişiere, nemaifiind nevoie să realizăm propriile noastre ferestre de dialog pentru aceste operaţiuni des întâlnite. Este vorba de un `JList` şi câteva butoane, plus alte componente care permit introducerea de date, toate legate împreună pentru a forma o singură componentă atomică numită `FileChooser` oferind aceeaşi funcţionalitate ca dialogurile fişier de pe platformele native. Pentru a seta directorul curent, se poate folosi metoda `setCurrentDirectory(String sir)`. Pentru a vizualiza un dialog, se pot utiliza apeluri de genul `showOpenDialog()` pentru deschiderea unui fişier sau `showSaveDialog()` pentru salvarea unui fişier.



Look In: JFileChooser

Name	Size	Type
Aplicatia.class	1 KB	CLASS File
Aplicatia	1 KB	Java Source File
b	1 KB	MS-DOS Batch File
deschide	1 KB	GIF Image

File Name: FereastraAplicatie.java

Files of Type: All Files

Componenta `JFileChooser`

Deoarece iniţializarea unei ferestre `FileChooser` necesită mult timp, este indicat ca operaţia de iniţializare (crearea unei instanţe a variabilei pentru dialog `JFileChooser dialog = new JFileChooser()`) să se realizeze într-un fir de execuţie separat la startarea aplicaţiei. `JFileChooser` permite folosirea de filtre pentru a vizualiza într-un `FileChooser` numai anumite tipuri de fişiere. Pentru aceasta, se extinde clasa `FileFilter` din pachetul `javax.swing.filechooser`, din care se suprascriu două metode:

Metoda	Explicaţie
`boolean` **`accept`**`(File f)`	Returnează `true`, dacă se doreşte vizualizarea fişierului `f`, altfel se va returna `false`.
`String` **`getDescription`**`()`	Returnează o descriere a filtrului folosit în `JcomboBox`-ul din partea de jos a dialogului `JFileChooser`.

Pentru a gestiona filtrele, `JFileChooser` pune la dispoziţie câteva metode:

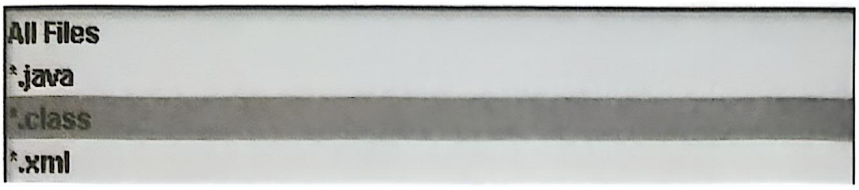
Metoda	Explicaţie
`addChoosableFileFilter` `(FileFilter f)`	Pentru a adăuga un alt filtru.
`removeChoosableFileFilter` `(FileFilter f)`	Pentru a şterge un filtru deja existent.
`setFileFilter` `(FileFilter f)`	Pentru a seta un filtru să fie activ (şi îl adaugă, dacă este necesar).

Implicit, `JFileChooser` foloseşte un filtru care permite vizualizarea tuturor fişierelor (engl. *all files*). Putem să ştergem acest filtru folosind secvenţa de comenzi:

```
FileFilter ft = dialog.getAcceptAllFileFilter();
dialog.removeChoosableFileFilter(ft);
```

Prezentăm o clasă şablon care poate fi simplu folosită pentru a implementa propriile noastre filtre:

```
class Filtru extends javax.swing.filechooser.FileFilter
  {
    private String descriere = null;
    private String extensie = null;

    public Filtru(String extensie, String descriere) {
      this.descriere = descriere;
      this.extensie = "." + extensie.toLowerCase();
    }

    public String getDescription() {
      return descriere;
    }

    public boolean accept(File f) {
      if (f == null)
        return false;
      if (f.isDirectory())
        return true;
      return f.getName().toLowerCase().endsWith(extensie);
    }
  }
```

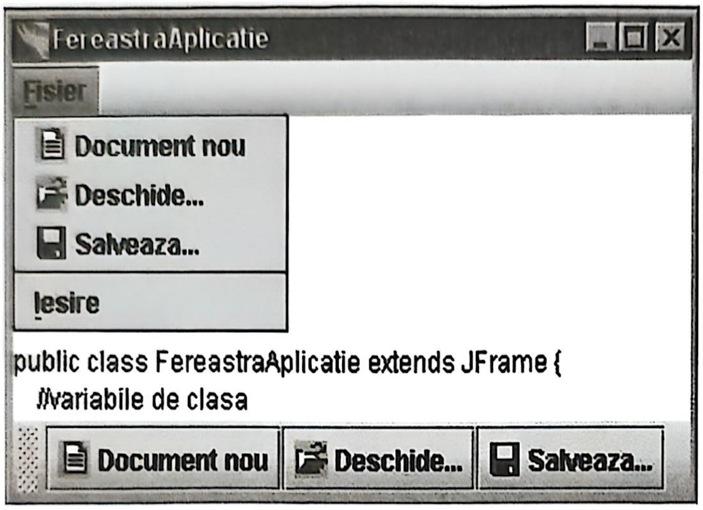

All Files
*.java
*.class
*.xml

Alegerea extensiei pentru fişiere

```
Dialogul.setFileFilter(new Filtru("java","*.java"));
Dialogul.setFileFilter(new Filtru("class","*.class"));
Dialogul.setFileFilter(new Filtru("xml","*.xml"));
```

A se observa că filtrul implicit nu a fost şters.

Prezentăm în continuare o implementare a meniului `Fisier` (engl. *file*), întâlnit în toate aplicaţiile. În acelaşi timp se adaugă şi o bară `JToolBar` ferestrei de bază.


FereastraAplicatie
Fisier
Document nou
Deschide...
Salveaza...
Iesire
Document nou
Deschide...
Salveaza...

```
public class FereastraAplicatie extends JFrame {
  //variabile de clasa
```

Implementarea meniului `Fisier`

```
protected void doMeniu() {
  meniuBara = new JMenuBar(); // cream bara de meniu
  meniuBara.setOpaque(true);
  JMenu Fisier= doMeniuFisier(); //cream un meniu
  // optiuni
  Fisier.setMnemonic('F'); // definim shortcut (Alt+F)
  meniuBara.add(Fisier); // adaugam meniul optiuni la
  // bara de meniu
  setJMenuBar(meniuBara);
}
// construim meniul optiuni
protected JMenu doMeniuFisier() {

  final JFileChooser Dialogul = new JFileChooser();
  Dialogul.setCurrentDirectory(new File("."));

  JMenu meniuFisier= new JMenu("Fisier");

  ImageIcon iconNou = new ImageIcon("nou.gif");
```

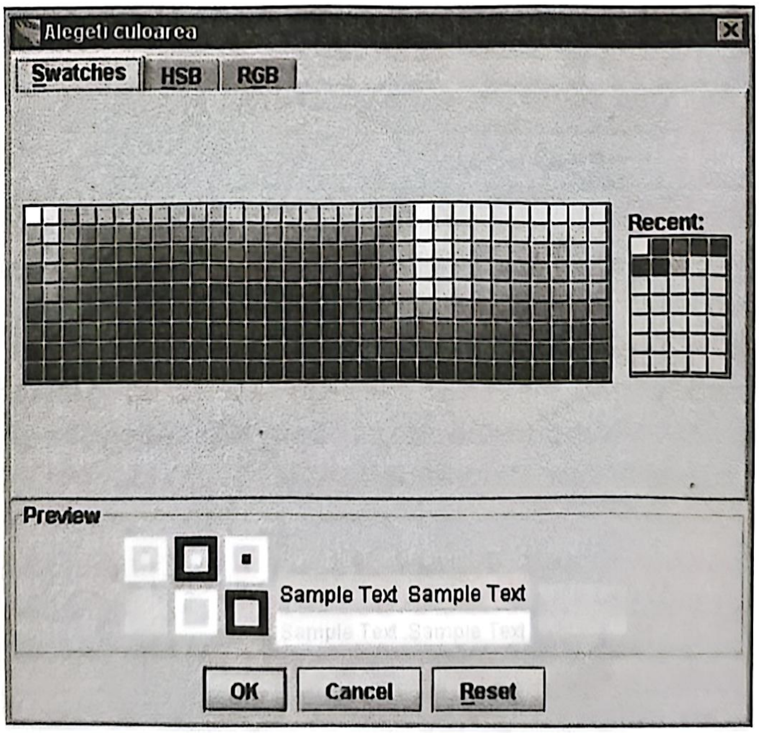
```
Action actiuneNou =
  new AbstractAction("Document nou", iconNou)
 {
   public void actionPerformed(ActionEvent e)
   {
     zonaText.setText("");
   }
 };
JMenuItem element =  meniuFisier.add(actiuneNou);
meniuFisier.add(element);

ImageIcon iconDeschide = new ImageIcon("deschide.gif");
Action  actiuneDeschide= new AbstractAction
  ("Deschide...", iconDeschide)
{
   public void actionPerformed(ActionEvent e)
    {
     if (Dialogul.showOpenDialog(FereastraAplicatie.this)
        != JFileChooser.APPROVE_OPTION)
        return;
     Thread fir = new Thread() {
       public void run() {
         File fisier = Dialogul.getSelectedFile();
         try
         {
          FileReader intrare = new FileReader(fisier);
          zonaText.read(intrare, null);
          intrare.close();
         }
         catch (IOException ex)
         {
          ex.printStackTrace();
         }
     }};
     fir.start();
    }
 };
element =  meniuFisier.add(actiuneDeschide);
meniuFisier.add(element);

ImageIcon iconSalveaza = new ImageIcon("salveaza.gif");
Action actiuneSalveaza = new AbstractAction(
  "Salveaza...", iconSalveaza)
  {
   public void actionPerformed(ActionEvent e)
{
 if (Dialogul.showSaveDialog(FereastraAplicatie.this)
      != JFileChooser.APPROVE_OPTION)
      return;
      Thread fir = new Thread() {
        public void run() {
          File fisier = Dialogul.getSelectedFile();
          try
          {
            FileWriter iesire = new FileWriter(fisier);
            zonaText.write(iesire);
            iesire.close();
          }
          catch (IOException ex)
          {
           ex.printStackTrace();
          }
      }};
      fir.start();
     }
};
element =  meniuFisier.add(actiuneSalveaza);
meniuFisier.add(element);

meniuFisier.addSeparator();

Action actiuneIesire = new AbstractAction("Iesire")
{
 public void actionPerformed(ActionEvent e)
 {
  System.exit(0);
 }
};

element =  meniuFisier.add(actiuneIesire);
element.setMnemonic('I');

JToolBar baraUnelte = new JToolBar();
JButton bNew = new JButton(actiuneNou);
baraUnelte.add(bNew);

JButton bOpen = new JButton(actiuneDeschide);
baraUnelte.add(bOpen);

JButton bSave = new JButton(actiuneSalveaza);
```

```
        baraUnelte.add(bSave);

        FereastraAplicatie.this.getContentPane().add(baraUnelte,
          BorderLayout.NORTH);

        return meniuFisier;
}
```

13.9.3. JColorChooser

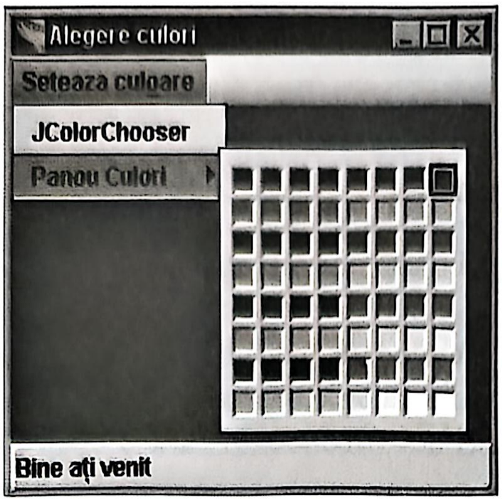
`JColorChooser` reprezintă o componentă standard pusă la dispoziţia dezvoltatorilor şi folosită pentru selecţia unei culori. Fereastra `JColorChooser` este folosită modal (blochează execuţia aplicaţiei). Permite trei modalităţi de alegere a culorilor `Swatches`, `HSB` şi `RGB`. Putem vedea şi alege din culorile deja selectate. O fereastră de dialog se activează folosind constructorul clasei `JColorChooser`, având drept parametri fereastra-părinte, numele ferestrei, respectiv culoarea implicită prezentată la deschiderea dialogului.

```
Color culoare = JColorChooser.showDialog(this,"Alegeti culoarea",
   Color.red);
if (culoare != null){
// actiune dupa alegerea culorii
continut.setBackground(culoare);
// in acest caz, coloreaza background-ul folosind
// culoarea aleasa
}
```


Alegeti culoarea
Swatches
HSB
RGB
Recent:
Preview
Sample Text Sample Text
Sample Text Sample Text
OK
Cancel
Reset

Componenta `JColorChooser`

Modul în care arată fereastra de dialog `JColorChooser` poate fi personalizat, dar nu vom insista asupra acestui lucru, vizualizarea acestui dialog părând a fi suficientă. Vom prezenta, în schimb, o metodă de a selecta culorile folosind un panou deschis la alegerea unei opțiuni dintr-un meniu.



Meniu pentru alegerea culorii

```
class PaletaCulori extends JMenu
{
  protected Border neselectat;
  protected Border selectat;
  protected Border activ;

  protected Hashtable perechi;
  protected PanouCuloare casuta;

  public PaletaCulori(String nume) {
   super(nume);
   neselectat = new CompoundBorder( new MatteBorder(1, 1, 1, 1,
    getBackground()), new BevelBorder(BevelBorder.LOWERED,
    Color.white, Color.gray));
   selectat = new CompoundBorder(  new MatteBorder(2, 2, 2, 2,
    Color.red), new MatteBorder(1, 1, 1, 1, getBackground()));
   activ = new CompoundBorder( new MatteBorder(2, 2, 2, 2,
     Color.blue), new MatteBorder(1, 1, 1, 1, getBackground()));

   JPanel panou = new JPanel();
   panou.setBorder(new EmptyBorder(5, 5, 5, 5));
   panou.setLayout(new GridLayout(8, 8));
   perechi = new Hashtable();

   int[] values = new int[] { 0, 128, 192, 255 };
   for (int r=0; r<values.length; r++) {
      for (int g=0; g<values.length; g++) {
```

```
      for (int b=0; b<values.length; b++) {
      Color c = new Color(values[r], values[g], values[b]);
      PanouCuloare pn = new PanouCuloare(c);
      panou.add(pn);
      perechi.put(c, pn);
      }
    }
  }
  add(panou);
}

public void seteazaCuloare(Color c) {
  Object obj = perechi.get(c);
  if (obj == null)
    return;
  if (casuta != null)
  casuta.setSelected(false);
  casuta = (PanouCuloare)obj;
  casuta.setSelected(true);
}

public Color daCuloare() {
  if (casuta == null)
    return null;
  return casuta.daCuloare();
}

public void selecteaza() {
  // actiune dorita
  FereastraAplicatie.continut.setBackground(daCuloare());
}

class PanouCuloare extends JPanel implements MouseListener
{
  protected Color m_c;
  protected boolean casuta;

  public PanouCuloare(Color c) {
    m_c = c;
    setBackground(c);
    setBorder(neselectat);
    String msg = "R "+c.getRed()+", G "+c.getGreen()+
      ", B "+c.getBlue();
    setToolTipText(msg);
    addMouseListener(this);
    }

    public Color daCuloare() { return m_c; }

    public Dimension getPreferredSize() {
      return new Dimension(15, 15);
    }
    public Dimension getMaximumSize() {
      return getPreferredSize(); }
    public Dimension getMinimumSize() {
      return getPreferredSize(); }

    public void setSelected(boolean selected) {
      casuta = selected;
      if (casuta)
        setBorder(selectat);
      else
        setBorder(neselectat);
    }

    public boolean isSelected() { return casuta; }

    public void mousePressed(MouseEvent e) {}

    public void mouseClicked(MouseEvent e) {}

    public void mouseReleased(MouseEvent e) {
      seteazaCuloare(m_c);
      MenuSelectionManager.defaultManager().
        clearSelectedPath();
      selecteaza();
    }

    public void mouseEntered(MouseEvent e) {
      setBorder(activ);
    }

    public void mouseExited(MouseEvent e) {
      setBorder(casuta ? selectat :  neselectat);
    }
  }
}
```

13.10. Componente pentru progres şi derulare

`JSlider`, `JScrollBar` şi `JProgressBar` sunt componente care ilustrează un domeniu interval şi o valoare ce evoluează între o valoare minimă şi o valoare maximă reprezentând cele două capete ale intervalului. Interfaţa `BoundedRangeModel` reprezintă un model de date care defineşte o valoare întreagă, ce reprezintă un indicator evoluând între o valoare minimă şi una maximă. Această valoare are asociată o alta care reprezintă mărimea indicatorului. Valoarea nu poate fi niciodată mai mică decât valoarea minimă şi mai mare decât valoarea maximă.

`DefaultBoundedRangeModel` reprezintă implementarea implicită a interfeţei `BoundedRangeModel`. Acest model aruncă evenimente `ChangeEvents` conform specificaţiei *JavaBeans*, atunci când o proprietate îşi schimbă valoarea. Toate cele trei componente descrise în continuare au implicit asociat drept model de date o instanţă a clasei `DefaultBoundedRangeModel`.

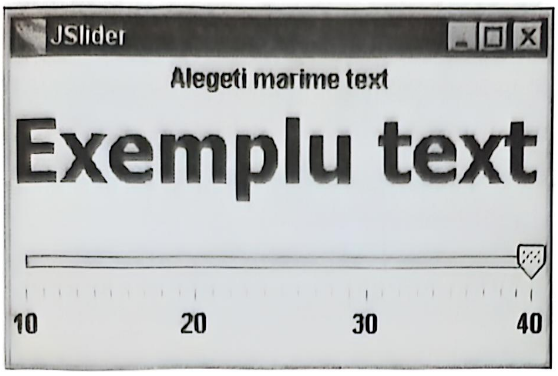
13.10.1. `JSlider`

`JSlider` reprezintă o metodă convenabilă pentru a alege o valoare numerică prin mutarea unui indicator pe suprafaţa unei rigle marcate. Componenta `JSlider` este foarte folositoare în cazul în care se cunoaşte domeniul de valori pe care trebuie să le introducă utilizatorul, permiţând acestuia alegerea facilă a uneia dintre valorile posibile. Valorile sunt evidenţiate pe riglă utilizând marcaje mici sau mari şi cresc de la stânga spre dreapta şi de jos în sus. Dacă poziţia marcajelor mari corespunde cu a celor mici, acestea din urmă nu vor fi afişate. Marcajele mari pot fi întotdeauna etichetate folosind componente, iar implicit acestea sunt etichete `JLabel` care le prezintă valoarea.

Componenta `JSlider` are următoarele proprietăţi:

Proprietăţi	Explicaţii
`orientation`	Oferă posibilitatea de a orienta vertical sau orizontal componenta prin setarea acestei proprietăţi cu valorile `JSlider.HORIZONTAL` sau `JSlider.VERTICAL`.
`extent`	Specifică numărul de valori sărite înainte şi înapoi prin folosirea tastelor `PageUp` şi `PageDown`.
`minorTickSpacing`	Spaţiul dintre marcajele mari.
`majorTickSpacing`	Spaţiul dintre marcajele mici.
`paintTicks`	Indică dacă se vor desena marcaje.
`paintLabels`	Setează/desetează afişarea etichetelor cu valori pentru marcajele mari.
`paintTrack`	Specifică dacă o componentă este colorată în interior sau nu.
`inverted`	Schimbă direcţia incrementării.

Pentru fiecare proprietate avem metode de acces corespunzătoare. De fiecare dată când una dintre proprietăţi îşi schimbă starea, conform specificaţiei *JavaBeans* se va arunca un eveniment `PropertyChangeEvent` ce încorporează proprietatea a cărei valoare s-a schimbat, putând astfel trata evenimentul produs. A se vedea secţiunea 13.3.4, dedicată descrierii JavaBeans.



Componenta `JSlider`

```
static JTextArea text;
//...
protected void doContinut() throws Exception {
  JPanel continut= new JPanel(); // in acest panou vom
  // adauga continutul ferestrei
  this.getContentPane().add(continut,BorderLayout.CENTER);
  int dis=10;
  JLabel eticheta = new JLabel("Alegeti marime text",
    JLabel.CENTER);
  eticheta.setAlignmentX(Component.CENTER_ALIGNMENT);
  JSlider slider = new JSlider(JSlider.HORIZONTAL,10,
    40, dis);
  slider.addChangeListener(new AscultatorSlider());
  slider.setMajorTickSpacing(10);
  slider.setMinorTickSpacing(1);
  slider.setPaintTicks(true);
  slider.setPaintLabels(true);
  slider.setBorder(BorderFactory.createEmptyBorder
    (0,0,10,0));
  Font f=new Font("Tahoma", Font.BOLD, 10);
  text=new JTextArea();
  text.setFont(f);
  text.setForeground(new Color(128,32,128));
  text.setBackground(eticheta.getBackground());
  text.setText("Exemplu text");
  text.setEditable(false);
  continut.setLayout(new BorderLayout());
  continut.add(eticheta,BorderLayout.NORTH);
  continut.add(text,BorderLayout.CENTER);
```

```
    continut.add(slider,BorderLayout.SOUTH);
  }

  class AscultatorSlider implements ChangeListener {
    public void stateChanged(ChangeEvent e) {
      JSlider sursa = (JSlider)e.getSource();
       if (!sursa.getValueIsAdjusting()) {
          int fps = (int)sursa.getValue();
          Font f=new Font("Tahoma", Font.BOLD, fps);
          FereastraAplicatie.text.setFont(f);
          FereastraAplicatie.text.repaint();
       }
    }
  }
```

13.10.2. JScrollBar

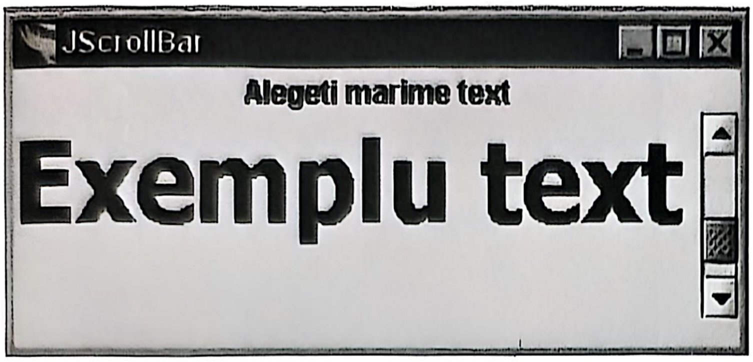
Reprezintă implementarea unei bare de derulare. Această componentă este des utilizată ca parte integrantă a componentei `JScrollPane` în procesul de derulare a conținutului, dar poate fi folosită și separat, așa cum vom vedea în această secțiune. Utilizatorul „plimbă" un indicator între două limite, una inferioară și una superioară (de fapt, alege o valoare dintr-un interval domeniu). Pentru acest lucru pot folosi și butoanele care se află la cele două extreme ale componentei. Zona componentei `ScrollBar`, care se află în exteriorul indicatorului și al butoanelor, poartă numele de zona de paginare și poate fi folosită la rândul ei pentru a muta indicatorul prin poziționarea cursorului mouse-ului. Deosebirea ei față de `JSlider` (aparent au aceeași funcționalitate) constă în faptul că `JSlider` este utilizată pentru a permite utilizatorului să aleagă o valoare.

Indicatorul are o anumită mărime, care este gestionată prin intermediul proprietății `visibleAmount`. Folosind metoda `setOrientation()` putem seta orientarea componentei recurgând cele două constante pe care le avem la dispoziție `JScrollBar.HORIZONTAL` și `JScrollBar.VERTICAL`. Proprietatea `unitIncrement` specifică pasul cu care se va face deplasarea indicatorului prin apăsarea butoanelor extreme. De asemenea, pasul cu care se va muta indicatorul atunci când folosim zona de paginare este gestionat prin intermediul proprietății `blockIncrement`.

Constructorul acestei componente are prototipul `public JScrollBar(int orientare, int valoare, int indicator, int minim, int maxim)`, unde `orientare` reprezintă orientarea dată de cele două constante enumerate mai sus, `valoare` reprezintă valoarea inițială (locul unde se va poziționa inițial indicatorul), `minim` valoarea minimă, `maxim` valoarea maximă, iar `indicator` reprezintă mărimea indicatorului.

Evenimentul pe care îl aruncă componenta `JScrollBar`, atunci când schimbăm poziția indicatorului, este `AdjustmentEvent`. Tratarea acestui eveniment se face cu ascultătorul `AdjustmentListener`. A se vedea exemplul următor.

Proprietatea `visibleAmount` oferă posibilitatea setării dimensiunii indicatorului. Dacă dimensiunile ferestrelor se modifică, trebuie ca şi dimensiunile indicatorului să se modifice proporţional pentru a păstra un raport corect. În exemplul următor am suprascris metoda `paint()` a containerului de bază pentru a redesena fereastra având indicatorul componentei `JScrollBar`, remodificat la noile dimensiuni.



Componenta `JScrollBar`

```
JScrollBar derulare;
Dimension dim = this.getPreferredSize();
int i=dim.height;
// construim continut
protected void doContinut() throws Exception {
   JPanel continut = new JPanel(); // in acest panou vom
   // adauga continutul ferestrei
   this.getContentPane().add(continut,BorderLayout.CENTER);

   int dis = 10;
   JLabel eticheta = new JLabel("Alegeti marime text",
      JLabel.CENTER);
   eticheta.setAlignmentX(Component.CENTER_ALIGNMENT);

   derulare = new JScrollBar( JScrollBar.VERTICAL, 10,
      i/10, 10, 60);
   derulare.addAdjustmentListener(new AscultatorScroll());
   derulare.setUnitIncrement(1);
   derulare.setBorder(BorderFactory.createEmptyBorder
      (0,0,10,0));

    Font f = new Font("Tahoma", Font.BOLD, 10);
    text=new JTextArea();
    text.setFont(f);
    text.setForeground(new Color(128,32,128));
    text.setBackground(eticheta.getBackground());
    text.setText("Exemplu text");
    text.setEditable(false);
```

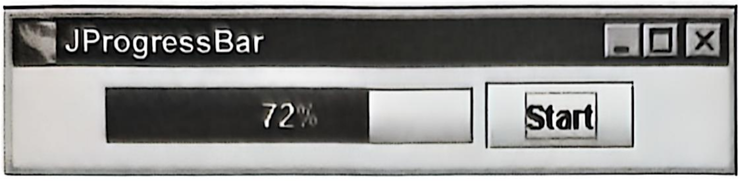
```
        continut.setLayout(new BorderLayout());
        continut.add(eticheta,BorderLayout.NORTH);
        continut.add(text,BorderLayout.CENTER);
        continut.add(derulare,BorderLayout.EAST);
    }

    public void paint(Graphics G)
    {
        Dimension dim = this.getPreferredSize();
        int i = dim.height;
        super.paint(G);
        derulare.setVisibleAmount(i/10);
    }


    class  AscultatorScroll implements AdjustmentListener {
      public void adjustmentValueChanged(AdjustmentEvent e){
        JScrollBar sursa = (JScrollBar)e.getSource();
        if (!sursa.getValueIsAdjusting()) {
          int fps = (int)sursa.getValue();
          Font f = new Font("Tahoma", Font.BOLD, fps);
          FereastraAplicatie.text.setFont(f);
          FereastraAplicatie.text.repaint();
    }}}
```

13.10.3. JProgressBar

Aşa cum îi spune şi numele, componenta `JProgressBar` este folosită pentru a indica progresul unei operaţii consumatoare de timp, arătând utilizatorului că operaţia monitorizată este activă şi în desfăşurare. Grafic se reprezintă printr-o bară orizontală sau verticală continuă (în cazul *look-and-feel*-ului Windows XP este formată din mici dreptunghiuri) care creşte dinspre stânga spre dreapta (sau de jos în sus), de la o valoare minimă spre una maximă. Prin intermediul acesteia poate fi vizualizat procentul din acţiune care a fost deja consumat.

Culoarea fundalului şi a *foreground*-ului se modifică la fel ca în cazul oricărei componente. Proprietatea `borderPainted` specifică dacă se va desena un chenar în jurul componentei. În timpul monitorizării unei activităţi vom folosi metoda `setValue()` pentru a reactualiza starea barei de progres. Această metodă trebuie utilizată în firul de execuţie *event-dispatching thread* pentru a nu apărea probleme la desenare.

`JProgressBar` a fost concepută pentru a monitoriza operaţii care durează un timp lung. Din punct de vedere computaţional, prin timp lung se înţelege mai mult de o secundă. O atenţie specială trebuie acordată logicii aplicaţiei prin care se actualizează grafic starea barei de progres. Există două moduri de a programa o bară de progres, şi anume modul analogic şi modul digital. `JProgressBar` poate fi folosită pentru a arăta că se petrece ceva şi nu pentru a măsura exact cât a evoluat o operaţiune. Acest mod de a vedea lucrurile poartă numele de mod analogic, în acest caz subliniindu-se mai mult progresele care au loc în desfăşurarea operaţiei, oferind astfel posibilitatea de a face comparaţii între cât s-a desfăşurat din operaţie şi ce mai este de făcut. Modul digital este folosit atunci când se doreşte reprezentarea exactă a modului în care decurge operaţia, progresul barei de progres fiind continuu şi fără salturi. În general, cele două moduri de prezentare se combină, deoarece deseori avem de a face cu operaţii care decurg greu, iar programatorul nu poate şti de la început când se vor termina.



Componenta `JProgressBar`

```
protected void doContinut() throws Exception
{
  final int m_min=1, m_max=100;

  JPanel continut= new JPanel();
  // in acest panou vom adauga
  // continutul ferestrei
  this.getContentPane().add(continut,BorderLayout.CENTER);
  UIManager.put("ProgressBar.selectionBackground",
       Color.black);
  UIManager.put("ProgressBar.selectionForeground",
       Color.white);
  UIManager.put("ProgressBar.foreground", new Color
       (128,32,128));

  final JProgressBar jpb = new JProgressBar();
  jpb.setMinimum(m_min);
  jpb.setMaximum(m_max);
  jpb.setStringPainted(true);

  JButton start = new JButton("Start");

  class AscultatorStart implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      Thread runner = new Thread() {
      int m_counter = m_min;
      public void run() {
            while (m_counter <= m_max) {
              Runnable runme = new Runnable() {
                public void run() {
                  jpb.setValue(m_counter);
                }
              };
              SwingUtilities.invokeLater(runme);
              m_counter++;
              try {
                Thread.sleep(100);
              }
              catch (Exception ex) {}
            }
          }
        };
        runner.start();
      }
    }

    AscultatorStart Start = new AscultatorStart();
    start.addActionListener(Start);
    continut.add(jpb, BorderLayout.CENTER);
    continut.add(start, BorderLayout.WEST);
    }
```

O altă metodă de a monitoriza operaţiile o reprezintă clasa `ProgressMonitor` care poate fi folosită foarte eficient pentru a monitoriza operaţii cu fişiere, având suport pentru controlul operaţiilor de scriere sau citire dinspre acestea. Pentru mai multe informaţii sunteţi invitaţi să consultaţi API-ul.

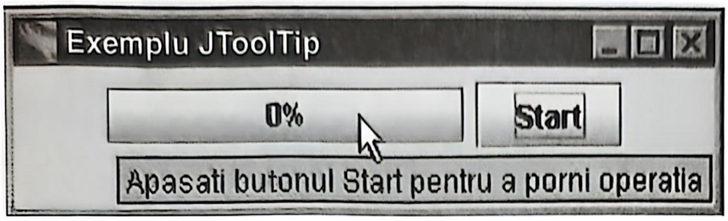
13.10.4. `JToolTip`

`JtoolTip`-urile reprezintă mici ferestre ce permit asocierea de informaţii textuale aplicaţiilor, vizualizate în momentul în care cursorul mouse-ului se află deasupra componentei. Acest mecanism este implementat în Swing, astfel încât nu este nevoie să creăm componente `JToolTip`. Este de ajuns să apelăm metoda `public void setToolTipText(String text)` moştenită de fiecare componentă din clasa `JComponent`. Prin acest apel, componenta este înregistrată de un gestionar implicit, instanţă a clasei `ToolTipManager`. De asemenea implicit, se va adăuga componentei un ascultător `MouseListener` pentru a determina când cursorul mouse-ului se va afla deasupra componentei. Pentru a anula `ToolTip`-ul asociat unei componente putem folosi un apel `setToolTipText(null)`.

Clasa `ToolTipManager` este cea care gestionează modul cum se va face prezentarea *tooltip*-urilor asociate componentelor. Atunci când cursorul mouse-ului intră în graniţele componentei, `ToolTipManager`-ul detectează acest lucru şi va aştepta

750 ms înainte de a afişa `ToolTip`-ul pentru componentă. Dacă cursorul rămâne deasupra componentei, `ToolTip`-ul va fi vizualizat timp de 4000 ms. Dacă în această perioadă de timp mutăm cursorul mouse-ului în afara componentei, *tooltip*-ul va dispărea. Dacă repoziţionăm cursorul mouse-ului deasupra componentei, `ToolTip`-ul asociat va apărea în 500 ms. Aceste trei perioade de timp pot fi setate folosind metodele puse la dispoziţie de clasa de serviciu `ToolTipManager`, şi anume: `setInitial Delay()`, `setDismissDelay()` şi `setReshowDelay()`.

Putem obţine o referinţă la instanţa `ToolTipManager` folosind un apel `ToolTip Manager toolTipManager = ToolTipManager.sharedInstance()`, putând astfel face modificări asupra managerului.



Componenta `JToolTip`

```
    // ...
    final JProgressBar jpb = new JProgressBar();
    JButton start = new JButton("Start");

    jpb.setToolTipText(
      "Apasati butonul Start pentru a porni operatia");
    start.setToolTipText("Porneste");
   ToolTipManager gestionar = ToolTipManager.sharedInstance();
    gestionar.setInitialDelay(0);
```

Personalizarea completă a modului în care se afişează *tooltip*-urile se realizează prin extinderi ale clasei `ToolTipManager`.

13.11. Ferestre interioare

Interfeţele pentru documente multiple (engl. *multiple document inteface* – MDI) reprezintă modul standard de a gestiona şi folosi mai multe documente pe aceeaşi suprafaţă de lucru, documente care sunt reprezentate de ferestre interioare. Motivul pentru care deseori se abordează această soluţie a unui *desktop* pe care se află fişiere este faptul că dacă am avea o fereastră `JFrame` de bază ce ar ocupa tot ecranul, în cazul în care ea deţine focusul, va ascunde toate celelalte ferestre şi dialoguri. În această situaţie, ar trebui să comutăm între ea şi aceste ferestre, lucru incomod. Din acest motiv, vom considera o fereastră care va acoperi în general ecranul, iar celelalte ferestre le vom considera interioare. Prezentăm în continuare cele două clase care sunt folosite în procesul de realizare a interfeţelor pentru documente multiple.

13.11.1. `JDesktopPane`

`JDesktopPane` este clasa care stă la baza realizării de interfeţe pentru documente multiple. Putem gândi `JDesktopPane` ca un container specializat pentru ferestre interioare. Fiecare `JDesktopPane` are asociată o implementare a interfeţei `DesktopManager`, care gestionează toate operaţiile efectuate de ferestrele `JInternalFrame`. Dacă nu se specifică explicit un astfel de gestionar, atunci se va asocia implicit o instanţă a clasei. Metodele acestui gestionar sunt automat apelate atunci când o acţiune este invocată de o fereastră `JInternalFrame` componentă (de fapt, o acţiune a utilizatorului asupra ferestrelor interioare). Câteva dintre aceste metode le vom prezenta în tabelul care urmează, însoţite de explicaţii :

Metoda	Explicaţia
`closeFrame``(JInternalFrame f)`	Închide fereastra.
`iconifyFrame``(JInternalFrame f)`	Iconifică fereastra.
`maximizeFrame``(JInternalFrame f)`	Maximizează fereastra.
`minimizeFrame``(JInternalFrame f)`	Minimizează fereastra.
`openFrame``(JIntenerlFrame f)`	Deschide fereastra.

Dacă dorim, spre exemplu, să minimizăm manual fereastra interioară `fereastraInt`, va trebui să realizăm un apel de forma `fereastraInt.getDesktopPane().getDesktopManager().iconifyFrame(fereastraInt)`. De asemenea, apelurile de metode asupra ferestrelor interioare, care se fac programatic, vor fi preluate de către gestionar pentru a fi executate. De remarcat faptul că `JDesktopPane` nu suportă chenarele.

13.11.2. `JInternalFrame`

Ferestrele interioare se aseamănă cu ferestrele obişnuite şi, chiar dacă le este gestionat asemănător conţinutul, prin intermediul unui panou `JRootPane`, ferestrele interioare nu sunt considerate containere de nivel înalt. Nu sunt componente „grele", neavând componente *peer* corespunzătoare, şi din acest motiv, sunt implementate independent de platformă. Ele sunt construite pentru a fi folosite în interiorul unei suprafeţe de lucru (engl. *desktop*) reprezentate de clasele `JDesktopPane` din secţiunea anterioară.

Proprietăţile unei ferestre interioare se pot controla mult mai uşor decât în cazul ferestrelor obişnuite. Ele se pot minimiza şi maximiza şi se poate specifica, de asemenea, pictograma unei ferestre. Se poate specifica dacă fereastra va fi decorată cu butoane pentru a suporta redimensionarea, transformarea în pictogramă, închiderea şi maximizarea.

O fereastră interioară poate fi creată folosind un constructor de forma:

```
public InternalFrame(String titlu, //titlul ferestrei
                     boolean redim, //redimensionare
```

```
                        boolean inchid,//inchidere
                        boolean maxim, //maximizare
                        boolean icon) ;//iconificare sau o forma
                        // supraincarcata a acesteia.
```

Pentru a adăuga o componentă unei ferestre interioare, aceasta se adaugă panoului ce gestionează conţinutul care se obţine printr-un apel `getContentPane()`. Pentru vizualizarea de dialoguri se vor folosi metodele statice de genul `showInternalXXX-Dialog()` ale clasei `JOptionPane`. Pentru a putea fi vizualizate ferestrele interioare trebuie adăugate unui container, în principiu, un `JDesktopPane`. Urmează vizualizarea ferestrei interne (în versiunile mai vechi ale platformei, acestea nu erau implicit invizibile) şi, eventual, selectarea ei pentru a deveni fereastră interioară activă.

După crearea unei ferestre interioare este nevoie să o dimensionăm, altfel ea va avea implicit dimensiunile nule şi nu va fi vizibilă. Pentru aceasta, se poate folosi una dintre metodele `setSize()`, `pack()` sau `setBounds()`. De asemenea, aceasta trebuie poziţionată folosind `setLocation()` sau `setBounds()`, altfel va fi vizualizată în colţul din stânga-sus al containerului său. Descriem o secvenţă de paşi care ilustrează cele discutate mai sus:

```
JDesktopPane suprafataLucru= new JDesktopPane();
JInternalFrame doc = new JInternalFrame(
  "Documentul nou",  true, true, true, true);
doc.setBounds(10,10,300,200);
suprafataLucru.add(doc);
doc.setVisible(true);
try {      doc.setSelected(true);     }
catch(Exception exe)
  { System.err.println("Exceptie");  }
this.getContentPane().add(suprafataLucru);
```

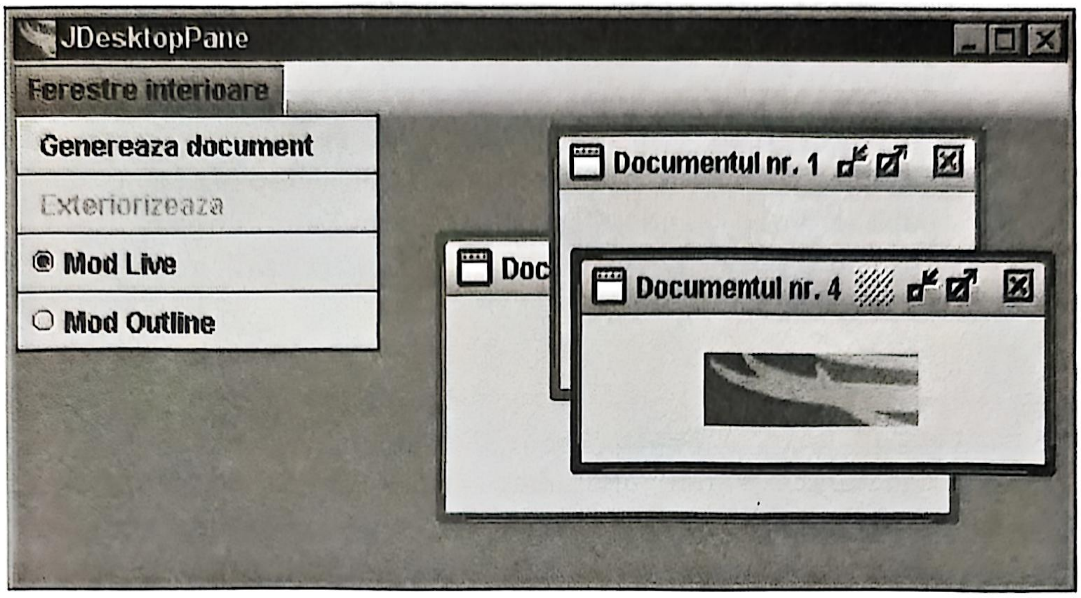
O fereastră `JInternalFrame` poate avea asociat unul sau mai mulţi ascultători `InternalFrameListener`, care tratează evenimentele `InternalFrameEvent` aruncate de fereastra interioară. Prezentăm, în continuare, câteva dintre metodele acestui ascultător, care trebuie suprascrise pentru a trata diverse situaţii în care evenimentul este aruncat:

Metoda	Explicaţii
`internalFrameClosed``(InternalFrameEvent e)`	Tratează închiderea ferestrei.
`internalFrameIconified``(InternalFrameEvent e)`	Tratează iconificarea ferestrei.
`internalFrameOpened``(InternalFrameEvent e)`	Tratează deschiderea ferestrei.

Pentru a nu suprascrie toate metodele avem la dispoziţie o extensie a acestui gestionar, şi anume `InternalFrameAdapter`. Ca optimizare, putem seta modul în care se va putea deplasa fereastra interioară cu mouse-ul, prin intermediul unei apel `setDragMode()` al clasei `JDesktopPane`, având drept parametru `JDesktopPane.OUTLINE_DRAG_MODE` pentru deplasare optimizată sau `LIVE_DRAG_MODE`, pentru deplasare standard.

Având o fereastră interioară, se poate pune problema exteriorizării ei, ceea ce înseamnă transformarea ei într-o fereastră obişnuită. Acest proces se realizează în câţiva paşi şi constă, de fapt, în trecerea conţinutului de la fereastra interioară, care va fi mai apoi închisă, spre o nouă fereastră obişnuită, proaspăt creată.

Fereastra interioară ce se doreşte exteriorizată se determină, eventual, printr-un apel al metodei `getSelected()` din clasa `JdesktopPane`, dacă ea este fereastra curent selectată.


JDesktopPane
Ferestre interioare
Genereaza document
Exteriorizeaza
● Mod Live
○ Mod Outline
Documentul nr. 1
Doc
Documentul nr. 4

Ferestre interne

```
/* Fereastra aplicatiei */
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.border.*;

public class FereastraAplicatie extends JFrame {
     //variabile de clasa
     static JMenuBar meniuBara;
     final Image img;
     int pozitie=1;
     JDesktopPane suprafataLucru= new JDesktopPane();
     static final Integer DOCLAYER = new Integer(5);
     int contor=0;

     //constructorul
     public FereastraAplicatie() throws Exception {
          super("JDesktopPane");
          // pozitionam fereastra si-i setam dimensiunile
          Toolkit t = this.getToolkit();
          Dimension marimeEcran = Toolkit.getDefaultToolkit().
          getScreenSize();
```

```
        setBounds ( pozitie, pozitie, marimeEcran.width
        pozitie, marimeEcran.height20*pozitie);
        // setam pictograma din coltul stanga-sus al ferestrei
        img = t.getImage("icon.jpg");
        this.setIconImage(img);
        this.getContentPane().setLayout(new BorderLayout());
        // construim partile componente
        doMeniu();
        doContinut();
        // specificam modul de reactie a ferestrei la inchidere
        this.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
        iesire();
        }});
    }

    protected void doMeniu() {
        meniuBara = new JMenuBar(); // cream bara de meniu
        meniuBara.setOpaque(true);
        JMenu optiuni= doMeniuOptiuni(); //cream un meniu
        //optiuni
        optiuni.setMnemonic('M'); // definim shortcut (Alt+M)
        meniuBara.add(optiuni); // adaugam meniul optiuni la
        // bara de meniu
        setJMenuBar(meniuBara);
    }

      // construim meniul optiuni
      protected JMenu doMeniuOptiuni() {
      JMenu unelte= new JMenu("Ferestre interioare");
      JMenuItem gen = new JMenuItem("Genereaza document");
      final JMenuItem ext = new JMenuItem("Exteriorizeaza");
      ext.setEnabled(false);

      gen.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            JInternalFrame doc = new JInternalFrame("Documentul nr."
              +(++contor),
              true, //redimensionabila
              true, //se poate inchide
              true, //se poate minimiza
              true);
            doc.getContentPane().add(new JLabel(new ImageIcon
            (img)));
```

```
doc.addInternalFrameListener(new
    InternalFrameAdapter()
{
  public void internalFrameClosed(InternalFrameEvent e)
     {
     ext.setEnabled(suprafataLucru.
     getSelectedFrame()!=null? true:false);
     }
});

doc.setBounds(10*contor,10*contor,300,200);
suprafataLucru.add(doc);
doc.setVisible(true);
try {
  doc.setSelected(true);
  ext.setEnabled(suprafataLucru.getSelectedFrame()!=
     null? true:false);
}
catch(Exception exe)
     { System.err.println("Exceptie");}
repaint();
}});

 unelte.add(gen);
 unelte.addSeparator();

 unelte.add(ext);
 unelte.addSeparator();
 ext.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
     JInternalFrame ferAct=
        suprafataLucru.getSelectedFrame();
     JFrame ferNoua=new JFrame();
     if (ferAct!=null)
     {
        ferAct.setVisible(false);
        ferNoua.setContentPane(ferAct.getContentPane());
     }
     ferAct.disable();
     ext.setEnabled(false);
     ferNoua.setBounds(10*contor,10*contor,290,190);
     ferNoua.setVisible(true);
   }});

 JRadioButtonMenuItem o1 =
           new JRadioButtonMenuItem("Mod Live");
        o1.setSelected(true);
        JRadioButtonMenuItem o2 =
           new JRadioButtonMenuItem("Mod Outline");
        // adaugam ascultatorul pentru optiunea1
        ButtonGroup grup=new ButtonGroup();
        grup.add(o1);
        grup.add(o2);

        o1.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e)  {
           suprafataLucru.setDragMode(
             JDesktopPane.LIVE_DRAG_MODE);
        }});
        // adaugam ascultatorul pentru optiunea2
        o2.addActionListener(new ActionListener() {
          public void actionPerformed(ActionEvent e)  {
          suprafataLucru.setDragMode(
             JDesktopPane.OUTLINE_DRAG_MODE);
          }});
          // adaugam optiunile la meniu
          unelte.add(o1);
          unelte.addSeparator();
          unelte.add(o2);
          return unelte;
    }

    // construim continut
    protected void doContinut() throws Exception
    {
          this.getContentPane().add(suprafataLucru);
    }

    public void iesire() {
        System.exit(0); // parasim aplicatia
      }
}
```

13.12. Recomandări şi optimizări

Ca o primă observaţie, se recomandă despărţirea părţii logice a aplicaţiei (engl. ***business logic***) de interfaţa grafică, din motive de reutilizare a codului, atât pentru interfaţă, cât şi pentru partea de logică a aplicaţiei. Astfel, se poate interveni în oricare parte a aplicaţiei, fără a o afecta pe cealaltă, iar partea logică poate fi folosită şi prin intermediul altor interfeţe grafice (aparţinând altor aplicaţii). Acest aspect poate fi benefic în cazul aplicaţiilor distribuite, în care partea de logică a aplicaţiei se poate găsi pe o altă maşină, aflată la distanţă. Despărţirea se face prin construirea a două clase diferite, una pentru logică şi alta pentru interfaţă, interfaţa realizând interogări asupra părţii logice a aplicaţiei, ca urmare a interacţiunii cu utilizatorii.

De asemenea, folosirea de denumiri sugestive pentru numele componentelor reprezintă un lucru benefic, care se va observa pe parcursul dezvoltării aplicaţiei, chiar dacă pare mai simplu să folosim în grabă nume scurte şi fără înţeles. Spre exemplu, dacă dăm nume sugestive mai multor componente folosind expresii regulate, prin intermediul *Reflection API* putem obţine foarte uşor referinţe despre acele componente. De asemenea, este indicată folosirea de comentarii care să ne ajute mai apoi în descoperirea greşelilor. O scriere elegantă a codului aduce beneficii în procesul de depanare şi de menţinere a aplicaţiei.

Cel mai important motiv pentru care interfeţele grafice merg greoi îl reprezintă nefolosirea eficientă a firelor de execuţie. Chiar dacă în Swing şi desenarea, şi expedierea evenimentelor se realizează într-un singur fir de execuţie, pentru optimizarea codului trebuie exploatate la maximum firele de execuţie şi metodele `invokeAndWait()`, respectiv `invokeLater()` (a se vedea secţiunea dedicată firelor de execuţie în Swing).

Un alt motiv de încetinire în execuţia aplicaţiei îl reprezintă crearea componentelor prin constructori care nu folosesc modele explicite, ci modele implicite. Adăugarea de elemente direct componentelor prin metodele pe care acestea le pun la dispoziţie determină aruncarea de evenimente inutile, care îngreunează aplicaţia. De aceea, este recomandată crearea de modele proprii, care vor fi folosite mai apoi în procesul de creare a componentelor. Înţelegerea arhitecturii MVC poate îmbunătăţi considerabil timpii aplicaţiei. Pentru aceasta, trebuie să parcurgeţi partea introductivă a acestui capitol.

13.13. Concluzii

Acest capitol prezintă biblioteca Swing, care pune la dispoziţia dezvoltatorilor de aplicaţii Java componentele necesare pentru realizarea interfeţelor grafice.

Capitolul debutează cu o introducere în JFC şi cu o descriere a componentelor Swing şi a câtorva aspecte fundamentale legate de acestea. De asemenea, sunt prezentate toate pachetele care alcătuiesc biblioteca Swing.

Urmează principii care stau la baza bibliotecii Swing, cum ar fi modelul de structurare MVC, modelul *UI-Delegate*, *look-and-feel*-ul aplicaţiilor, prezentarea bibliotecii Swing în raport cu AWT. În cazul *look-and-feel*-ului, am precizat referinţe spre pagini de unde cititorii pot să le descarce.

Subcapitolul „Fundamentele Swing" prezintă detaliat sistemul de evenimente şi ascultători, firele de execuţie în Swing, desenarea componentelor, concordanţa cu JavaBeans, respectiv gestiunea focusului. Subcapitolul se încheie cu o clasificare a componentelor ce determină structura subcapitolelor următoare, în care componentele vor fi prezentate detaliat, grupate după cum se va vedea în continuare.

Secţiunea „Containere de bază" prezintă ferestrele Swing şi clasele şi interfeţele care au legătură cu acestea.

Secţiunea „Componente atomice simple" prezintă componentele etichete, butoane, chenare, liste.

Subcapitolul „Componente atomice complexe" prezintă componentele text, tabele şi arbori.

Am detaliat apoi în subcapitole separate meniurile şi barele de unelte, dialogurile, componentele pentru gestiunea progresului şi pentru derulare şi ferestrele interioare.

Ultima secţiune cuprinde recomandări pentru dezvoltatori şi optimizări anunţate pentru viitoarele distribuţii ale platformei Java.

13.14. Test-grilă

Întrebarea 13.14.1. Ce reprezintă Swing?

a) O interfaţă prietenoasă cu utilizatorii.
b) O bibliotecă de componente grafice.
c) O componentă grafică.
d) O metodă de a realiza efecte de animaţie.

Întrebarea 13.14.2. Care este deosebirea între Swing şi AWT?

a) Componentele Swing sunt scrise în totalitate în Java, pe când în AWT componentele sunt scrise folosind cod nativ.
b) Componentele AWT au vizualizarea dependentă de sistemul de operare, iar, în Swing, componentele pot avea aceeaşi vizualizare, indiferent de sistemul de operare.
c) Componentele Swing sunt mai uşoare decât componentele AWT.

Întrebarea 13.14.3. Care dintre afirmaţiile următoare sunt adevărate?

a) Fiecare componentă AWT are o componentă corespondentă în Swing.
b) Componentele Swing au denumirea componentelor AWT, la care se adaugă prefixul „J".
c) O componentă este un obiect cu reprezentare grafică.

Întrebarea 13.14.4. Cum se realizează vizualizarea unei componente grafice?

a) Vizualizarea componentelor grafice se realizează odată cu apelarea constructorului.
b) Vizualizarea componentelor grafice se realizează prin apelarea metodei `setVisible()`.
c) Vizualizarea componentelor grafice se realizează prin adăugarea componentei la un container deja vizibil.
d) Vizualizarea componentelor grafice se realizează printr-un apel `setContainerLayout(null)`.

Întrebarea 13.14.5. Ce efect va avea pentru un container de bază apelul metodei `setBounds(10,10,100,200)`?

a) Componenta va avea acum lăţimea 90 şi înălţimea 190.
b) Componenta se va poziţiona la punctul având coordonatele colţului din stânga-sus (10, 10) şi coordonatele colţului din dreapta-jos (100, 200).
c) Componenta va avea lăţimea 100 şi înălţimea 200.

Întrebarea 13.14.6. Ce reprezintă *pluggable look-and-feel*-ul pentru o componentă grafică?

a) Posibilitatea de a ascunde componenta în momentul rulării interfeţei.
b) O metodă de a schimba aspectul componentei şi al modului în care aceasta interacţionează cu utilizatorii.
c) O metodă standard de a adăuga *plug-in*-uri aplicaţiilor Java.

Întrebarea 13.14.7. Care dintre afirmaţiile următoare sunt corecte?

a) Un model de date păstrează datele unei componente grafice.
b) Toate componentele Swing au modele.
c) Modelele deţin informaţii despre vizualizarea tabelei.
d) Modelele interacţionează cu vizualizarea doar prin intermediul evenimentelor.

Întrebarea 13.14.8. Fie secvenţa de cod:

```
JButton b=new JButton();
public class TratareIesire implements ActionListener {
     public void actionPerformed(ActionEvent e) {
         dispose(); // inchide fereastra curenta
     }
}
ActionListener[] al=(ActionListener[])(
      b.getListeners(ActionListener.class));
if(al.length==0)
      b.addActionListener(new TratareIesire());
```

Ce reprezintă secvenţa de cod precedentă?

a) Adăugarea mai multor ascultători unui buton.
b) Adăugarea unui ascultător unui buton, numai în cazul în care acesta nu mai are un altul asociat.
c) Ştergerea unui ascultător.

Întrebarea 13.14.9. Care dintre afirmaţiile următoare sunt adevărate?

a) Unei componente îi putem adăuga mai mulţi ascultători de acelaşi tip.
b) Un adaptor (engl. *adapter*) este o clasă abstractă care extinde o interfaţă şi pe care o moştenim pentru a nu fi nevoiţi să implementăm fiecare metodă a interfeţei respective.
c) `KeyEvent` este evenimentul generat de apăsarea unui buton al mouse-ului.
d) `MouseListener` este ascultătorul care permite captarea evenimentelor generate de mouse.

Întrebarea 13.14.10. Care dintre afirmaţiile următoare sunt adevărate?

a) Swing nu permite folosirea altor fire de execuţie diferite de *event-despatching thread.*
b) Dacă un container este redesenat, toate componentele conţinute sunt redesenate şi ele.
c) Metoda *double-buffering* dă posibilitatea desenării direct pe ecran a componentelor.
d) Swing permite folosirea a numai patru fonturi.
e) Focusul reprezintă fluxul acces spre dispozitivul standard de intrare.

Întrebarea 13.14.11. Care dintre afirmaţiile următoare sunt adevărate?

a) `JPanel` este un container de bază;
b) `JRootPane` este container intermediar;
c) `JApplet` reprezintă un applet.

Întrebarea 13.14.12. Cum se adaugă un buton unui container de bază `JFrame`?

a) `JFrame f = new JFrame(); JPanel continut = (JPanel)f.getContentPane(); continut.add(new JButton("OK"));`
b) `JFrame f = new JFrame(); f.getContentPane().add(new JButton ("OK"));`
c) `JFrame f = new JFrame(); f.add(new JButton("OK"));`

Întrebarea 13.14.13. Care dintre afirmaţiile următoare sunt adevărate?

a) Panoul `ContentPane` este gestionat implicit de `FlowLayout`.
b) Orice alt panou diferit de `ContentPane` este gestionat implicit de `FlowLayout`.
c) Apelul `setLayout(null)` determină ca în containerul curent poziţionarea să se facă absolut (relativ la colţul din stânga-sus al ecranului).

Întrebarea 13.14.14. Cum se face ca un arbore `JTree` să poată fi derulat de componenta `JScrollPane`?

a) `JScrollPane sp = new JScrollPane(new JTree(model));`
b) `JScrollPane sp = new JScrollPane(); sp.add(new JTree(model));`
c) `JScrollPane sp = new JScrollPane();`
`sp.setViewportView(new JTree(model));`

Întrebarea 13.14.15. Care dintre afirmaţiile următoare sunt adevărate?

a) `JTabbedPane` permite crearea unui panou format din alte două subpanouri alăturate.
b) Panourile care alcătuiesc un `JSplitPane` respectă dimensiunile minime ale componentelor.
c) `JTabbedPane` reprezintă o stivă de componente dintre care numai una este vizibilă la un moment dat.

Întrebarea 13.14.16. Cum se face ca o tabelă `JTable` să poată fi derulată de componenta `JScrollPane`?

a) `JScrollPane sp = new JScrollPane(new JTable(model));`

b) `JScrollPane sp = new JScrollPane(); sp.add(new JTable(model));`

c) `JScrollPane sp = new JScrollPane(); sp.add(new JTable(model)); sp.setScrollable(true);`

Întrebarea 13.14.17. Fie secvenţa de cod:

```
JPanel p = new JPanel();
JLabel e = new JLabel("Eticheta:");
e.setDisplayedMnemonic('E');
p.add(et, BorderLayout.EAST);
JTextField t = new JTextField(7);
e.setLabelFor(t);
p.add(t, BorderLayout.WEST);
```

Ce se va întâmpla prin apăsarea combinaţiei *Alt+E*?

a) Eticheta `e` va deţine focusul.

b) Câmpul `t` va deţine focusul.

c) Nici o componentă nu va mai deţine focusul.

Întrebarea 13.14.18. Care dintre ascultătorii următori indică bifarea unui `JCheckBox`?

a) `ActionListener`

b) `EventListener`

c) `ItemListener`

d) `MouseListener`

Întrebarea 13.14.19. Care dintre afirmaţiile următoare sunt adevărate?

a) Un obiect de tip chenar (engl. *border*) este o componentă grafică.

b) Un chenar `TitledBorder` permite asocierea unui titlu unui grup de componente simple.

c) Un *tooltip* reprezintă o bară de unelte.

Întrebarea 13.14.20. Care dintre afirmaţiile următoare sunt adevărate?

a) La un moment dat, într-o listă poate fi selectat un singur element.

b) `JComboBox` reprezintă o listă expandabilă.

c) Formatarea conţinutului pentru `JList` şi `JComboBox` se face prin intermediul clasei `ListCellRenderer`.

d) Componenta `JSpinner` are un model de date numit `JSpinnerListModel`, care permite setarea facilă a datei calendaristice.

Întrebarea 13.14.21. Care dintre următoarele afirmaţii este adevărată?

a) Secvenţa de cod care urmează va arunca o excepţie:

```
JPasswordField p = new JPasswordField();
String text = p.getText();
```

b) Metodele `copy()` şi `cut()` folosesc *clipboard*-ul sistemului de operare şi, de aceea, printr-o operaţiune `paste()` executată dintr-o altă aplicaţie, eventual nu Java, va alipi conţinutul.

c) Pentru a face ca o componentă text să nu poată obţine focusul într-un ciclu focus, se va apela metoda `transferFocus()`.

d) Modelul `PlainDocument` ne furnizează prin suprascriere posibilitatea de a personaliza componentele text.

e) Swing nu oferă suport pentru operaţiile *undo/redo*.

Întrebarea 13.14.22. Care dintre afirmaţiile următoare sunt adevărate?

a) Întotdeauna o componentă `JTable` are asociat un model de date, chiar dacă nu specificăm explicit acest lucru.

b) Prin modificări efectuate asupra datelor din model, se va actualiza automat şi vizualizarea.

c) Pentru a crea o tabelă folosind modelul `AbstractTableModel`, este de ajuns să suprascriem metodele `getRowCount()` şi `getColumnCount()`.

d) Modelul de date `AbstractTableModel` ţine datele întotdeauna într-un `Vector` având elemente `Vector`.

e) O tabelă permite derularea conţinutului fără a trebui inclusă într-un panou `JScrollPane`.

Întrebarea 13.14.23. Fie secvenţa de cod:

```
class  Copie extends AbstractAction
{
  Copie(String nume, ImageIcon icon)
      { super(nume, icon); }
  public void actionPerformed(ActionEvent e)
      { text.copy();    }
}
JMenuBar m = new JMenuBar();
JMenu e = new JMenu("Editare");
JMenuItem c = e.add(new Copie("Copie",new ImageIcon
  ("copie.gif")));
m.add(e);
```

Ce se va întâmpla la execuţia codului?

a) Se va arunca o excepţie care marchează faptul că nu se poate adăuga un eveniment unui meniu.

b) Se va crea un meniu-bară, cu un submeniu *Editează*, având o opţiune *Copie*.

c) Se va arunca o excepţie, pentru că opţiunile meniurilor Java nu pot avea pictograme.

Întrebarea 13.14.24. Care dintre afirmaţiile următoare sunt adevărate?

a) Clasa `JOptionPane` permite crearea de dialoguri folosită numai împreună cu clasa `JDialog`.

b) Fie secvenţa de cod:

```
Object[] optiuni = {"Da!", "Nu!"};
```

```
int nr = JOptionPane.showOptionDialog(fereastra, "Alegeti optiune",
  "Mesaj cu optiuni personalizate", JOptionPane.YES_NO_OPTION,
  JOptionPane.QUESTION_MESSAGE, null, optiuni, optiuni[0]);
```

Dacă vom apăsa butonul „`Da!`”, `nr` va lua valoarea `1`.

c) În cazul în care folosim componenta `JFileChooser`, prin alegerea unui nume de fişier, în aplicaţie vom obţine o referinţă `File` spre acesta.

d) Fereastra `JColorChooser` blochează întotdeauna celelalte ferestre ale aplicaţiei.

Întrebarea 13.14.25. Care dintre afirmaţiile următoare sunt adevărate?

a) `DefaultBoundedRangeModel` reprezintă modelul de date pentru clasele `JSlider`, `JScrollBar`, `JProgressBar`.

b) `JScrollBar` este folosită de componenta `JScrollPane` pentru a derula conţinutul.

c) În cazul componentei `JProgressBar`, modificarea poziţiei indicatorului se realizează prin apeluri `moveTo()`.

Întrebarea 13.14.26. Ce determină secvenţa de cod?

```
JDesktopPane suprafataLucru = new JDesktopPane();
suprafataLucru.setDragMode(JDesktopPane.OUTLINE_DRAG_MODE);
```

a) Ferestrele interioare conţinute în panoul `suprafataLucru` nu vor mai putea fi mutate.

b) Ferestrele interioare conţinute în panoul `suprafataLucru` vor putea fi exteriorizate.

c) Ferestrele interioare conţinute în panoul `suprafataLucru` vor putea fi mutate cu un consum mai mic de resurse.

13.15. Exerciţii propuse spre implementare

Exerciţiul 13.15.1. Realizaţi animaţii folosind API-ul Java2D.

Exerciţiul 13.15.2. Testaţi modul în care se vor comporta componentele `JTable` şi `JTree` în cazul unor volume mari de date. Gândiţi-vă şi încercaţi diverse optimizări.

Exerciţiul 13.15.3. Realizaţi o aplicaţie care monitorizează toţi ascultătorii din maşina Java curentă şi evenimentele pe care aceştia le ascultă.

Exerciţiul 13.15.4. Testaţi efectele pe care le au firele de execuţie utilizator ce modifică interfaţa Swing pentru o aplicaţie oarecare. Remediaţi deficienţele folosind metodele indicate în secţiunea dedicată firelor de execuţie.

13.16. Proiecte propuse spre implementare

Proiectul 13.16.1. Realizaţi un editor vizual pentru documentele de tip XML, folosind componenta `JTable`. Se va evita folosirea componentelor `JTree` şi a componentelor text.

Proiectul 13.16.2. Realizaţi un convertor XSL, având interfaţă grafică prietenoasă. Pentru a permite editarea, se va folosi clasa `JEditorPane`, despre care găsiţi informaţii în API-ul Java.

Proiectul 13.16.3. Creaţi un *look-and-feel* pentru Java, eventual pornind de la un *look-and-feel* deja existent.

Proiectul 13.16.4. Creaţi un administrator al resurselor maşinii virtuale Java. Acesta va avea o interfaţă grafică atractivă şi va permite curăţarea memoriei, apelând *garbage collector*, va arăta consumul de memorie etc.

Proiectul 13.16.5. Realizaţi un convertor care transformă aplicaţiile grafice ce folosesc AWT în aplicaţii grafice folosind Swing. În tutorialul Java există explicaţii legate de o astfel de conversie.

Proiectul 13.16.6. Realizaţi un mediu vizual pentru Java respectând convenţiile din specificaţia JavaBeans. Acesta va avea o structură modulară, permiţând un nucleu de funcţionalităţi la care se vor putea adăuga în timp şi altele.

Proiectul 13.16.7. Realizaţi un browser Web folosind Swing. Acesta va putea vizualiza şi sursele paginilor vizitate.

Proiectul 13.16.8. Realizaţi un editor HTML folosind principiile MDI (*multiple document interface*). Pentru mai multe informaţii, a se vedea secţiunea „Ferestre interioare”.

Proiectul 13.16.9. Realizaţi un gestionar de fişiere (echivalentul aplicaţiei *Total Commander* pentru Java).

Proiectul 13.16.10. Realizaţi o aplicaţie care să fie un editor şi browser WAP.

14. Sistemul help pentru Java

14.1. Cuvinte-cheie

- JavaHelp
- HelpSet
- help senzitiv la context

14.2. Convenţii

Vom folosi denumirea generică de *help* în sens de ajutor, asistenţă, netraducând-o în limba română. În principiu, înseamnă modalitatea prin care dezvoltatorul produsului informatic explică, prezintă utilizatorilor aplicaţia. Poate fi folosit şi cu sensul mai larg de documentaţie, caz în care nu ne referim neapărat la o aplicaţie.

Prin *sistem help* înţelegem modalitatea pe care un limbaj o oferă programatorilor pentru a ataşa help aplicaţiilor. Sistemul help pentru platforma Java poartă numele *JavaHelp*.

Prin *HelpSet* vom înţelege totalitatea fişierelor care participă la realizarea sistemului de help pentru o aplicaţie oarecare dezvoltată în limbajul Java, păstrând astfel denumirea întâlnită în specificaţie şi în ghidurile de utilizare.

14.3. Principii de bază

Aplicaţiile complexe necesită şi un mod de prezentare a documentaţiei aferente, deoarece este vitală înţelegerea de către utilizatori a modului în care pot interacţiona cu aplicaţia pentru a-şi atinge obiectivele. Java pune la dispoziţia dezvoltatorilor un mod standard şi uşor de folosit pentru realizarea documentaţiei aplicaţiilor pe care aceştia le creează. Sistemul help pentru Java este reglementat printr-o specificaţie gestionată de firma Sun Microsystems, de care trebuie să ţină seama toţi cei care dezvoltă sisteme help pentru acest limbaj. De asemenea, firma Sun pune la dispoziţia programatorilor şi o implementare realizată în Java a acestei specificaţii, deci un sistem help efectiv, numit *JavaHelp*, care face subiectul acestui capitol. Această implementare oferă o interfaţă simplă, prin care dezvoltatorii pot adăuga help aplicaţiilor, şi permite chiar personalizarea sistemului help pentru a se potrivi cu stilul adoptat în realizarea produsului informatic. JavaHelp permite adăugarea de help atât aplicaţiilor *stand-alone*, cât şi appleturilor sau componentelor JavaBeans. De asemenea, prin intermediul unui server de help se poate adăuga asistenţă şi aplicaţiilor scrise în alte limbaje decât Java.

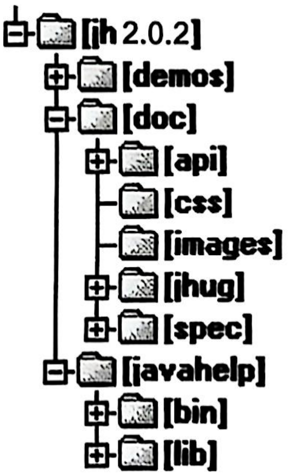
Folosind JavaHelp, autorul documentaţiei este responsabil pentru scrierea temelor (engl. *topics*) care tratează anumite subiecte într-un mod standard, şi anume în format HTML, precum şi a unei serii de metadate (informaţii despre date) structurate în fişiere XML. API-ul JavaHelp funcţionează ca un vizualizator care prezintă utilizatorului într-un mod standard (eventual personalizat) documentaţia alcătuită din multitudinea de teme, ţinând cont de modul în care acestea sunt structurate prin intermediul metadatelor. Putem spune că partea de structurare este despărţită de conţinut şi acestea sunt despărţite de modul vizualizare care este dat de API-ul JavaHelp. Astfel, cel care realizează documentaţia nu trebuie să ştie cum se procesează XML sau cum se foloseşte HTML în Java, ci se concentrează numai pe scrierea documentației și pe atașarea ei aplicației. Fișierele HTML împreună cu fișierele XML (chiar dacă uneori nu au extensia `.xml`) poartă numele de HelpSet, la fel ca și clasa de bază a API-ului, cea care face legătura dintre aplicație și documentație.

Pentru exemplificarea noțiunilor prezentate în acest capitol vom folosi drept referință aplicația JAIS, care gestiona colecțiile și fondurile arhivelor statului.

14.3.1. Prezentarea pachetului JavaHelp

Acest capitol se bazează pe implementarea implicită a sistemului help pentru Java, oferit de firma Sun, și anume JavaHelp, care reprezintă un pachet opțional pentru J2SDK. Acesta funcționează pe mai multe platforme, cu unele optimizări pentru a doua, optimizări posibile datorită dezvoltării și maturizării limbajului Java.

Ultima distribuție a sistemului JavaHelp, și anume *JavaHelp2.0.2* o puteți descărca gratuit de la adresa *http//java.sun.com/products/javahelp/*. După dezarhivarea pachetului se obține următoarea structură de directoare:



Structura de directoare a pachetului JavaHelp

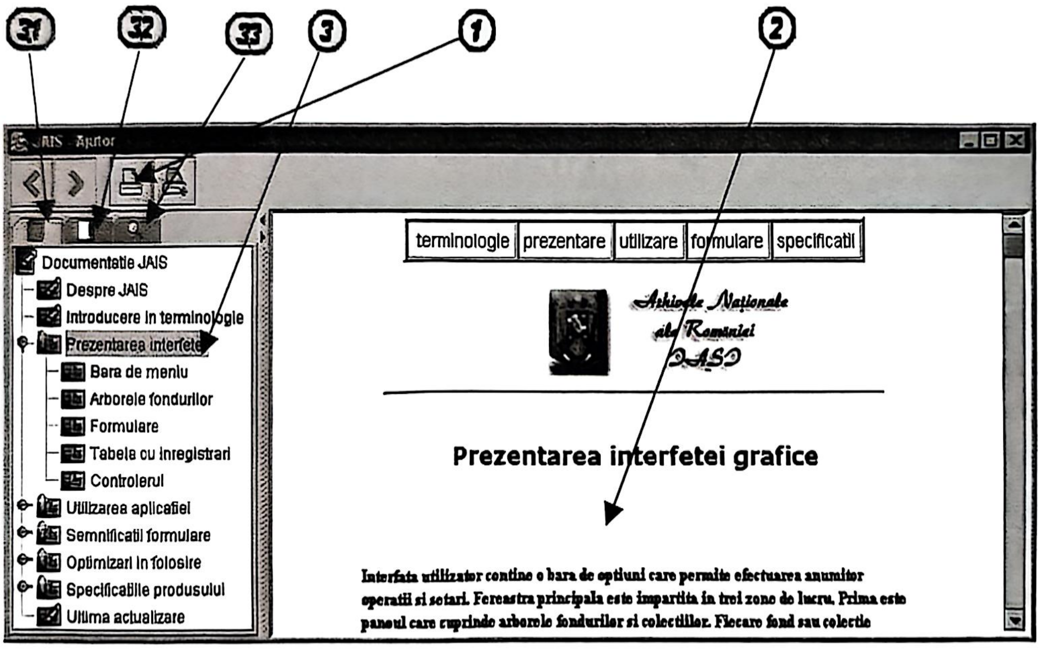
Pachetul JavaHelp vine și cu documentația aferentă, care se găsește în directorul `/doc` al distribuției. Aceasta este alcătuită din *javadoc*-ul care prezintă API-ul în forma standard bine cunoscută, specificația cu versiunea 2.0 a sistemului help pentru Java și un ghid de utilizare pentru această distribuție poziționat în directorul `/doc/jhug`. Ghidul (engl. *JavaHelp User Guide*) este prezentat chiar sub forma de JavaHelp, deci partea de HTML care stă la baza lui se poate vizualiza cu un browser Web oarecare. Același ghid de utilizare îl regăsim și în format PDF în chiar rădăcina pachetului. De asemenea, pachetul vine și cu o serie de exemple concrete de utilizare care se află în directorul `/demos`.

În directorul `javahelp/lib` se găsesc mai multe pachete care pot fi incluse în CLASSPATH-ul aplicației, fiecare dintre acestea având un anumit specific. Clasele sistemului JavaHelp sunt distribuite în mai multe fișiere JAR, utilizatorul incluzând în CLASSPATH arhiva corespunzătoare nevoilor aplicației pe care o dezvoltă:

Arhiva	Conținut
`jh.jar`	Reprezintă biblioteca standard şi include tot ceea ce este necesar pentru vizualizarea help-ului folosind modurile standard de navigare (cuprins, index, căutare).
`jhbasic.jar`	Un subset al `jh.jar` care nu cuprinde suport pentru căutări. Acesta poate fi folositor pentru sisteme simple de help care nu necesită căutări sau în cazul sistemelor în care spaţiul e important.
`jhall.jar`	Include toate clasele sistemului JavaHelp, inclusiv uneltele necesare pentru crearea bazei de date (indexului) necesare căutărilor.
`jsearch.jar`	Motorul de căutare implicit utilizat în sistemul JavaHelp.

Directorul `javahelp/bin` cuprinde o serie de instrumente standard necesare în procesul de dezvoltare a documentaţiei, şi anume `jhindexer`, care reprezintă motorul de indexare, şi `jhsearch`, reprezentând motorul de căutare.

14.3.2. Alcătuirea ferestrei standard JavaHelp


31
32
33
3
1
2
JAIS - Ajutor
Documentatie JAIS
Despre JAIS
Introducere in terminologie
Prezentarea interfetei
Bara de meniu
Arborele fondurilor
Formulare
Tabele cu inregistrari
Controlerul
Utilizarea aplicatiei
Semnificatii formulare
Optimizari in folosire
Specificatiile produsului
Ultima actualizare
terminologie
prezentare
utilizare
formulare
specificatii
Arhivele Nationale ale Romaniei IASI

Prezentarea interfetei grafice

Interfata utilizator contine o bara de optiuni care permite efectuarea anumitor operatii si setari. Fereastra principala este impartita in trei zone de lucru. Prima este panoul care cuprinde arborele fondurilor si colectiilor. Fiecare fond sau colectie

Fereastra standard JavaHelp

Interfaţa grafică standard a sistemului JavaHelp constă dintr-o bară de unelte şi două panouri:

1. *Bara de unelte* – permite navigarea înainte şi înapoi între temele documentaţiei la fel, ca într-un browser obişnuit, precum şi listarea la imprimantă a documentaţiei.
2. *Panoul conţinut* – prezintă helpul propriu-zis în formatul HTML plus unele elemente Java cum ar fi ferestre secundare şi ferestre *pop-up*. Conţinutul este prezentat sub formă de teme, de fapt, fişiere HTML care descriu un subiect ales în prealabil din panoul de navigare de către utilizatorul aplicaţiei.
3. *Panoul de navigare* – o interfaţă `tabbed` care permite utilizatorilor să comute între cuprins, index şi căutare, acestea reprezentând metodele standard puse la dispoziţie pentru navigarea în documentaţie, prezentate în continuare.

3.1. *Cuprinsul* este o reprezentare arborescentă a temelor constitutive ale sistemului help ataşat aplicaţiei, fiind, de fapt, cuprinsul documentaţiei. Suportă niveluri nelimitate şi alăturarea mai multor cuprinsuri corespunzătoare diferitelor HelpSet-uri. De fapt, această structură arborescentă este reprezentarea vizuală, folosind `JTree`, a unui document XML care stochează conţinutul documentaţiei. Vizualizarea cuprinsului este sincronizată cu panoul conţinut, numele temei vizualizate în conţinut fiind marcat în cuprins.

3.2. *Indexul* – tot o structură arborescentă, care reprezintă de această dată indexul documentaţiei. Suportă alăturarea mai multor indecşi. Ca şi în cazul cuprinsului, este vorba despre înfăşurarea unui fişier XML folosind JTree. La fel există sincronizare între panoul ce conţine indexul şi panoul conţinut.

3.3. *Căutarea* – un motor flexibil de căutare poate fi folosit într-o diversitate de medii de reţea. Numele temelor returnate în urma căutărilor sunt prezentate în mod descendent, în funcţie de potrivire.

14.3.3. Principii pe care sistemul JavaHelp le respectă

Sistemul JavaHelp respectă următoarele principii:

- *Compresia şi încapsularea* – formatul standard JAR poate fi folosit pentru a încapsula informaţiile help într-un singur fişier comprimat. Sistemul JavaHelp lucrează în mod egal cu informaţii help care nu sunt comprimate în fişiere JAR, această facilitate permiţând dezvoltatorilor să aibă acces la fişiere în timpul creării helpului fără să piardă timp pentru a comprima fişierele componente.
- *Scufundarea ferestrelor help* – ferestrele Help (individuale sau combinate) pot fi scufundate direct în interfeţele grafice ale aplicaţiilor.
- *Help senzitiv la context* – help-ul poate fi activat din interiorul aplicaţiilor folosind diverse mecanisme cum ar fi apăsarea tastei F1.
- *Personalizare* – sistemul JavaHelp este proiectat pentru a permite o mare flexibilitate atât în ceea ce priveşte interfaţa utilizator, cât şi în ceea ce priveşte funcţionalitatea.
- *Combinări* – informaţii help din diferite surse pot fi combinate şi prezentate utilizatorului final.
- *Suport pentru JavaBeans* – API-ul JavaHelp permite componentelor JavaBeans să specifice informaţii help care pot fi prezentate utilizatorilor finali (eventual, combinate cu informaţii suplimentare).

14.4. Dezvoltarea efectivă a help-ului

Reamintim că HelpSet-ul reprezintă totalitatea fişierelor necesare creării sistemului JavaHelp pentru aplicaţia curentă. Prezentăm, în continuare, lista paşilor necesari a fi parcurşi pentru crearea HelpSet-ului:

1. Crearea temelor HTML (în general, în fişiere separate) care formează împreună conţinutul documentaţiei.

2. Crearea fişierului HelpSet care leagă între ele toate metadatele.
3. Crearea fişierului Map care asociază identificatori temelor HTML.
4. Crearea fişierului cuprins care descrie conţinutul documentaţiei.
5. Crearea fişierului index care descrie indexul documentaţiei.
6. Crearea bazei de date necesare mai apoi în procesul de căutare.
7. Compresia şi încapsularea tuturor fişierelor într-o arhivă JAR.

Odată HelpSet-ul creat, pe parcursul procesului de dezvoltare puteţi să-l vizualizaţi, fără să-l ataşaţi aplicaţiei, folosind o unealtă inclusă în pachetul JavaHelp şi care este localizată în directorul `demos/bin/`, şi anume `hsviewer`. Sintaxa comenzii de apelare este:

```
java -jar hsviewer.jar [helpset nume_HS]
```

unde `numeHS` reprezintă fişierul HelpSet (cel cu extensia `.hs`). Dacă nu precizăm nici un fişier de intrare, îl veţi putea alege din interfaţa grafică ce va apărea.

Pentru a realiza toate operaţiile de la paşii precedenţi, puteţi folosi diverse unelte (engl. *tools*) puse la dispoziţie gratuit sau contra cost de diverse companii software. O listă a acestora o puteţi găsi în pagina de bază a produsului JavaHelp *http://java.sun.com/products/javahelp/industry.html*.

14.4.1. Temele HTML

Primul pas constă în scrierea conţinutului efectiv al documentaţiei, care este alcătuit dintr-o mulţime de fişiere HTML, în general, fiecare dintre ele descriind o anumită temă. De exemplu, pentru o aplicaţie, o temă poate descrie o funcţionalitate sau un element al interfeţei grafice. Între aceste fişiere HTML pot exista legături (engl. *links*) şi se recomandă folosirea lor în forma relativă, adică având drept referinţă directorul curent. Spre exemplu, legătura relativă de forma:

```
<a href="../prezentare/meniu.html">
  Prezentarea meniului</a>
```

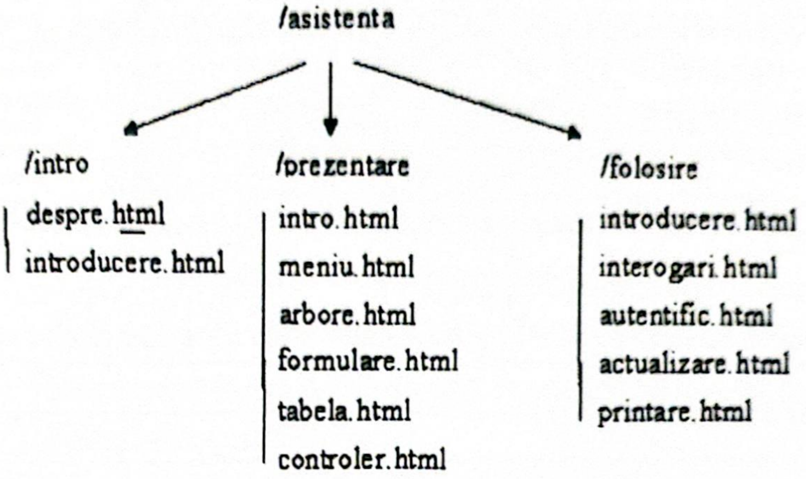
este de preferat legăturii absolute de forma:

```
<a href="C:/aplicatia/asistenta/prezentare/meniu.html">
  Prezentarea meniului</a>
```

deoarece legăturile relative rămân valide şi atunci când temele se arhivează sau se mută pe maşina utilizatorului. Separatorul între componentele URL-urilor trebuie să fie simbolul „/”.

Panoul din dreapta care vizualizează conţinutul se bazează pe componenta Swing `JEditorPane`. Din această cauză pot apărea probleme dintre care unele au fost fixate în timp sau pot fi evitate într-un fel sau altul (a se vedea ghidul de utilizare pentru amănunte).

Din experienţă, se recomandă să fie constituite cât mai multe fişiere (fiecare temă să fie stocată într-un fişier HTML separat), şi nu fişiere mari în care să se folosească ancore cu nume, din cauza modului greoi în care se face poziţionarea pe ancore. Procesul de deschidere a unui fişier mic este mai rapid decât poziţionarea pe o ancoră într-un fişier de mare dimensiune. O altfel de abordare este folositoare doar în cazurile în care toată documentaţia se poate restrânge la un singur fişier HTML împărţit în teme, care este utilizat pe toată durata cât aplicaţia este activă. În acest caz, fişierul este deschis o singură dată la început. Pentru a grupa totuşi temele după înţeles, se recomandă crearea unei ierarhii arborescente de directoare şi fişiere pe criterii semantice. Un bun exemplu ar putea fi următorul:


/asistenta
/intro
despre.html
introducere.html
/prezentare
intro.html
meniu.html
arbore.html
formulare.html
tabela.html
controler.html
/folosire
introducere.html
interogari.html
autentific.html
actualizare.html
printare.html

Organizarea documentelor HTML în directoare

14.4.2. Fişierul HelpSet

Atunci când sistemul JavaHelp este activat de către aplicaţie, primul lucru care se realizează este citirea fişierului HelpSet specificat de aplicaţie printr-o comandă de activare. Fişierul HelpSet defineşte pentru aplicaţie o mulţime de date care descriu sistemul help, unind de fapt toate celelalte metadate, constituindu-se în acest sens în fişierul de bază. Fişierul HelpSet include următoarele informaţii:

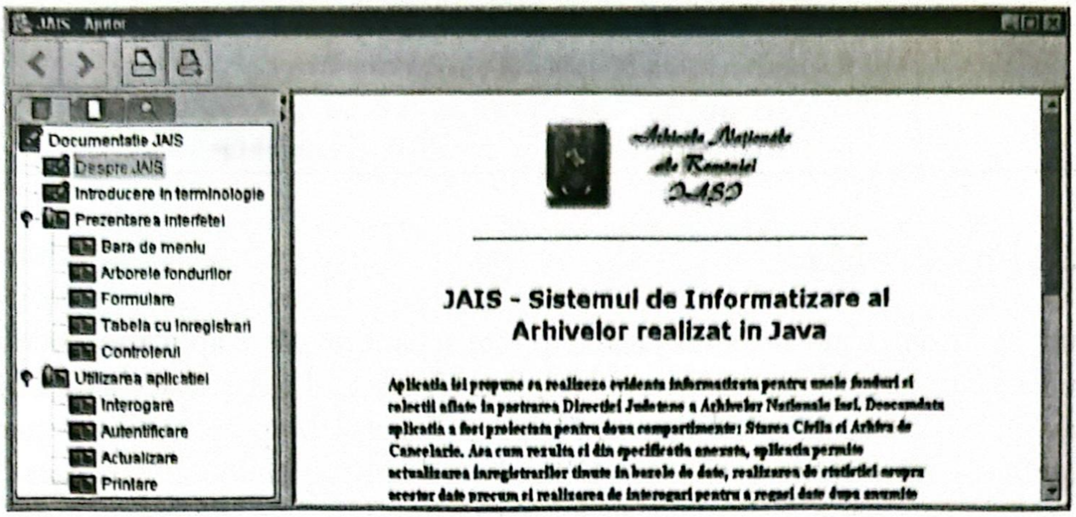
Informaţii	Descriere
fişierul Map	Fişierul Map este folosit pentru a asocia identificatori temelor (care sunt conţinute în fişiere HTML) folosind URL-uri (căi spre fişiere)
informaţii de vizualizare	Informaţii care descriu navigatoarele ce vor fi folosite de HelpSet. Navigatoarele standard sunt: cuprins, index, căutare. Informaţii despre navigatoarele personalizate sunt incluse, de asemenea, aici.
titlul HelpSet-ului	Numele directorului cel mai de sus din cuprins.
identificatorul Home	Numele identificatorului implicit care este vizualizat, atunci când vizualizatorul de help este chemat fără a specifica un ID în prealabil.
sub-HelpSet-uri	Secţiuni opţionale pot fi folosite pentru a include în mod static alte HelpSet-uri, folosind marcajul `<subhelpset>`. HelpSet-urile indicate de acest marcaj sunt adăugate în mod automat în HelpSet-urile care conţin marcajul.

După ce aplicaţia este instalată pe sistemul utilizatorului, pentru a se putea prezenta help-ul, trebuie să fie posibilă găsirea fişierului HelpSet. Aplicaţia foloseşte calea spre fişierul HelpSet, pentru a porni sistemul JavaHelp. Prin convenţie, numele fişierului HelpSet se termină cu extensia `.hs`. Formatul fişierului HelpSet se bazează pe XML.

În continuare prezentăm descrierea unui fişier HelpSet - `asistenta.hs`:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE helpset PUBLIC
 "-//Sun Microsystems Inc.//DTD JavaHelp HelpSet Version 1.0//EN"
 "http://java.sun.com/products/javahelp/helpset_1_0.dtd">
<helpset version="1.0">
<!-- titlul ferestrei  -->
<title>JAIS  Ajutor</title>
<!-- asocieri -->
<maps>
     <homeID>despre</homeID>
     <mapref location="mapare.jhm"/>
</maps>
<!-- vizualizari -->
<view>
     <name>Continut</name>
     <label>Continutul documentatiei</label>
     <type>javax.help.TOCView</type>
     <data>continut.xml</data>
</view>
<view>
     <name>Index</name>
     <label>Indexul documentatiei</label>
     <type>javax.help.IndexView</type>
     <data>index.xml</data>
</view>
<view>
     <name>Cautare</name>
     <label>Cautare in documentatie</label>
     <type>javax.help.SearchView</type>
     <data engine="com.sun.java.help.search.DefaultSearchEngine">
     JavaHelpSearch
     </data>
</view>
</helpset>
```

Acest fişier HelpSet determină vizualizarea următoarei ferestre JavaHelp:


Documentatie JAIS
Despre JAIS
Introducere in terminologie
Prezentarea interfetei
Bara de meniu
Arborele fondurilor
Formulare
Tabela cu inregistrari
Controlerii
Utilizarea aplicatiei
Interogare
Autentificare
Actualizare
Printare
Arhivele Nationale ale Romaniei IASI

JAIS - Sistemul de Informatizare al Arhivelor realizat in Java

Aplicatia isi propune sa realizeze evidenta informatizata pentru unele fonduri si colectii aflate in pastrarea Directiei Judetene a Arhivelor Nationale Iasi. Deocamdata aplicatia a fost proiectata pentru doua compartimente: Starea Civila si Arhiva de Cancelarie. Asa cum rezulta si din specificatia anexata, aplicatia permite actualizarea inregistrarilor tinute in bazele de date, realizarea de statistici asupra acestor date precum si realizarea de interogari pentru a regasi date dupa anumite

Fereastra de documentare a aplicației

Următoarea tabelă prezintă elementele care pot apărea într-un fişier HelpSet:

`<helpset>`	Elementul-rădăcină al HelpSet-ului. Poate conţine toate celelalte elemente.
`<title>`	Numele ferestrei HelpSet.
`<maps>`	Specifică tema implicită şi URL-ul fişierului Map folosit în HelpSet. Conţine următoarele elemente: • `<homeID>` Specifică numele identificatorului implicit a cărui temă asociată va fi afişată în panoul conţinut atunci când sistemul help este activat, dacă un identificator de temă nu este specificat explicit. • `<mapref>` Specifică fişierul Map. De remarcat că în distribuţia 1.0 se poate specifica doar un fişier Map. Conţine următorul atribut : - `location` URL-ul fişierului map.
`<view>`	Defineşte navigatoarele folosite în sistemul HelpSet. Poate conţine: • `<name>` Numele navigatorului. • `<label>` Specifică o etichetă asociată cu vizualizarea. Acest şir poate fi accesat de către aplicaţie şi folosit în prezentare. Este afişat la poziţionarea mouse-ului pe pictograma navigatorului. • `<type>` Specifică calea spre clasa navigatorului. • `<data>` Specifică calea spre datele folosite de navigator. Când este folosit de către navigatorul căutare, conţine atributul următor: - `engine` Calea spre motorul de căutare.

➔

➔

`<subhelpset>`	Acest tag opţional poate fi folosit pentru a specifica HelpSet-urile care vor fi combinate cu HelpSet-ul curent. `- location` URL-ul fişierului HelpSet care va fi combinat.

14.4.3. Fişierul Map

Aşa cum spuneam, când sistemul JavaHelp este activat de către aplicaţie, primul pas pe care îl va face este să citească fişierul HelpSet. Următorul pas este acela de a procesa fişierul Map localizat, folosind fişierul HelpSet. Fişierul Map este utilizat pentru a asocia temelor identificatori, folosind URL-uri (căi spre fişierele HTML care conţin temele). Prin convenţie, fişierele Map folosesc extensia `.jhm`. Formatul unui fişier Map se bazează, de asemenea, pe XML. Prezentăm, în continuare, un exemplu de fişier Map – `mapare.jhm`:

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<!DOCTYPE map PUBLIC
 "-//Sun Microsystems Inc.//DTD JavaHelp Map Version 1.0//EN"
 "http://java.sun.com/products/javahelp/map_1_0.dtd">
<map version="1.0">
  <mapID target="t.img" url="imagini/t.gif" />
  <mapID target="d.img" url="imagini/d.gif" />
  <mapID target="f.img" url="imagini/f.gif" />
  <mapID target="e.img" url="imagini/e.gif" />
  <mapID target="despre" url="intro/despre.html" />
  <mapID target="introducere.terminologie"
         url="intro/introducere.html" />
  <mapID target="prezentare.introducere"
         url="prezentare/intro.html" />
  <mapID target="prezentare.meniu"
         url="prezentare/meniu.html" />
  <mapID target="prezentare.arbore"
         url="prezentare/arbore.html" />
  <mapID target="prezentare.formulare"
         url="prezentare/formulare.html" />
  <mapID target="prezentare.tabela"
         url="prezentare/tabela.html" />
  <mapID target="prezentare.controler"
         url="prezentare/controler.html" />
  <mapID target="folosire.introducere"
         url="folosire/introducere.html" />
  <mapID target="folosire.interogari"
         url="folosire/interogari.html" />
  <mapID target="folosire.autentificare"
         url="folosire/autentificare.html" />
```

```
    <mapID target="folosire.actualizare"
           url="folosire/actualizare.html " />
    <mapID target="folosire.printare"
           url="folosire/printare.html " />
</map>
```

De notat că imaginile folosite (de exemplu, în cuprins) sunt, de asemenea, identificate folosind identificatori. Spre exemplu, fişierului `t.gif` i-am asociat identificatorul `t.img`.

În continuare sunt descrise elementele folosite în fişierele Map:

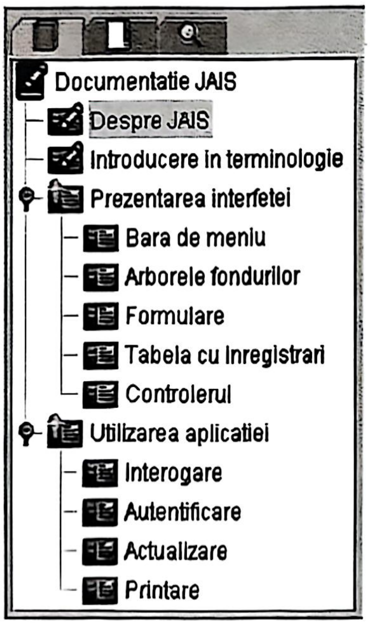
`<map>`	Elementul-rădăcină al fişierului Map. Poate conţine elemente `<mapID>`.
`<mapID>`	Defineşte o asociere de identificator unei teme. Are următoarele atribute: - `target` Specifică identificatorul asociat cu URL-ul specificat în atributul `url`. - `url` Specifică URL-ul asociat cu identificatorul specificat în atributul `target`.

14.4.4. Fişierul cuprins

Fişierul cuprins (engl. *Table Of Contents* – TOC) descrie conţinutul documentaţiei. Formatul unui fişier cuprins este bazat pe XML şi are extensia `.xml`. Urmează un scurt exemplu de fişier cuprins – `continut.xml`:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE toc PUBLIC
 "-//Sun Microsystems Inc.//DTD JavaHelp TOC Version 1.0//EN"
 "http://java.sun.com/products/javahelp/toc_1_0.dtd">
<toc version="1.0">
  <tocitem image="t.jpg" text="Documentatie JAIS"
           target="prima.pagina">
  <tocitem image="e.jpg"
           text="Despre JAIS" target="despre" />
  <tocitem image="e.jpg"
           text="Introducere in terminologie"
           target="introducere.terminologie" />
  <tocitem image="d.jpg"
           text="Prezentarea interfetei grafice"
           target="prezentare.introducere">
  <tocitem image="f.jpg"
           text="Bara de meniu"
           target="prezentare.meniu" />
  <tocitem image="f.jpg"
           text="Arborele fondurilor"
           target="prezentare.arbore" />
  <tocitem image="f.jpg"
           text="Formulare"
           target="prezentare.formulare" />
  <tocitem image="f.jpg"
           text="Tabela cu inregistrari"
           target="prezentare.tabela" />
  <tocitem image="f.jpg"
           text="Controlerul"
           target="prezentare.controler" />
  </tocitem>
  <tocitem image="d.jpg"
           text="Utilizarea aplicatiei"
           target="folosire.introducere">
  <tocitem image="f.jpg" text="Interogare"
           target="folosire.interogari" />
  <tocitem image="f.jpg" text="Autentificare"
           target="folosire.autentificare" />
  <tocitem image="f.jpg" text="Actualizare"
           target="folosire.actualizare" />
  <tocitem image="f.jpg" text="Printare"
           target="folosire.printare" />
  </tocitem>
  </tocitem>
</toc>
```

Acest exemplu determină vizualizarea următorului conţinut:



- Documentatie JAIS
  - Despre JAIS
  - Introducere in terminologie
  - Prezentarea interfetei
    - Bara de meniu
    - Arborele fondurilor
    - Formulare
    - Tabela cu inregistrari
    - Controlerul
  - Utilizarea aplicatiei
    - Interogare
    - Autentificare
    - Actualizare
    - Printare

Cuprinsul documentaţiei

Tabela următoare descrie elementele care apar în fişierele TOC:

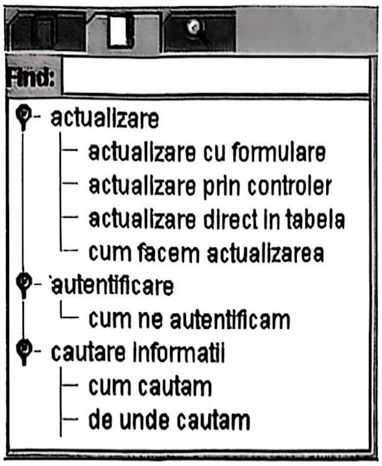
`<toc>`	Elementul-rădăcină al fişierului cuprins. Conţine subelementele `<tocitem>`.
`<tocitem>`	Defineşte o intrare în cuprins. Imbricarea `intrarea1` în `intrarea2` defineşte `intrarea2` ca fiind inclusă ierarhic în `intrarea1`. Se pot folosi următoarele atribute: – `text` Specifică numele intrării. – `target` (opţional) Specifică identificatorul asociat temei care va fi vizualizată, când această intrare va fi aleasă de utilizator. Identificatorii sunt definiţi în fişierul Map. – `image` (opţional) Specifică imaginea care va fi afişată în faţa intrării din conţinut. Argumentul acestui atribut este un identificator definit (asociat cu o imagine GIF sau JPEG) în fişierul Map. Dacă acest atribut nu este specificat, sunt afişate imaginile implicite pentru fişier şi director.

14.4.5. Fişierul index

Fişierul index descrie conţinutul navigatorului index şi modul în care va arăta acesta. Formatul unui fişier index se bazează pe XML. În continuare, prezentăm un exemplu simplu de fişier index – `index.xml`:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE index PUBLIC
 "-//Sun Microsystems Inc.//DTD JavaHelp Index Version 1.0//EN"
 "http://java.sun.com/products/javahelp/index_1_0.dtd">
<index version="1.0">
  <indexitem text="actualizare">
    <indexitem text="actualizare cu formulare"
               target="prezentare.formulare" />
    <indexitem text="actualizare prin controler"
               target="prezentare.controler" />
    <indexitem text="actualizare direct în tabela"
               target="prezentare.tabela" />
    <indexitem text="cum facem actualizarea"
               target="folosire.actualizare" />
  </indexitem>
  <indexitem text="autentificare">
    <indexitem text="cum ne autentificam"
               target="folosire.autentificare" />
  </indexitem>
  <indexitem text="cautare informatii">
    <indexitem text="cum cautam"
               target="folosire.interogari" />
    <indexitem text="de unde cautam"
               target="prezentare.formulare" />
  </indexitem>
</index>
```

Acest exemplu produce următoarea vizualizare:



- actualizare
  - actualizare cu formulare
  - actualizare prin controler
  - actualizare direct in tabela
  - cum facem actualizarea
- autentificare
  - cum ne autentificam
- cautare informatii
  - cum cautam
  - de unde cautam

Indexul documentației

Elementele folosite în fişierul index sunt următoarele:

`<index>`	Elementul-rădăcină. Poate conține subelementele `<indexitem>`.
`<indexitem>`	Defineşte o intrare în index. Imbricând `intrare1` în `intrare2` definim `intrare2` ca fiind ierarhic conținută de `intrare1`. Se folosesc următoarele atribute: - `text` Specifică textul intrării. - `target` (opțional) Specifică identificatorul temei ce va fi vizualizată, când intrarea este aleasă de utilizator. Identificatorii sunt definiți în fişierul Map.

14.4.6. Indexarea şi căutarea

Motorul de căutare al sistemului JavaHelp utilizează o tehnologie de căutare în limbaj natural care regăseşte acele pasaje din documentele constituente pentru care răspunsurile la interogări sunt corecte. Tehnologia implică un motor de indexare care analizează documentele pentru a produce un index al conținutului lor şi un motor de interogare care utilizează acest index pentru a regăsi pasaje relevante din material. Autorul helpului este cel care creează baza de date pentru căutare, ce va fi explorată de motorul de căutare al sistemului JavaHelp. Pentru crearea bazei de date se va folosi comanda `jhindexer` din directorul `javahelp/bin` al pachetului JavaHelp, care are următoarea sintaxă:

```
jhindexer [optiuni] [fisier | director]*
```

În cazul în care se specifică un director, se vor căuta în interior recursiv toate fişierele conţinând teme şi se vor indexa, iar dacă se specifică un fişier, el se va indexa.

Prezentăm în continuare lista opţiunilor posibile:

Opţiune	Explicaţia parametrilor
`-c fisier`	Comportarea comenzii poate fi controlată folosind un fişier de configurare; `fisier` reprezintă numele fişierului de configurare.
`-db director`	Parametrul `director` reprezintă directorul în care se va depune baza de date rezultată. Implicit, acesta este numit `JavaHelpSearch` şi este creat în directorul curent.
`-locale limba_tara`	Constanta `limba_tara` asociată localizării (limba şi ţara) determină caracteristica `locale` a indexului aşa cum apare în `java.util.Locale`. Pentru exemplu: `ro_RO` (romāna, România).
`-logfile fisier`	Capturează în fişierul `fisier` mesajele generate de `hindexer`.
`-nostopwords`	Cuvintele exceptate (engl. *stopwords*) vor fi indexate şi ele.
`-verbose`	Afişare de mesaje în timpul indexării.

Comanda:

```
java -jar c:/jh2.0/javahelp/bin/jhindexer.jar -c configurare.txt
    -locale ro_RO -logfile rezultat.txt -verbose
    intro prezentare folosire
```

executată în directorul rădăcină al sistemului help determină crearea bazei de date (indexului) formate din şase fişiere în directorul implicit `JavaHelpSearch`, folosind fişierul de configurare `configurare.txt`, rezultând de asemenea un fişier de jurnalizare pentru operaţia de indexare `rezultat.txt`. Ultimii trei parametri reprezintă numele directoarelor unde se găsesc temele HTML.

Pentru verificarea integrităţii bazei de date create, se foloseşte comanda `jhsearch directorul_indexului`, comandă care se află tot în directorul `javahelp/bin`.

Descriem în continuare posibilităţile pe care le oferă fişierul de configurare.

- memorarea numelor căii spre fişiere cu alt nume în baza de date, opţiune importantă în cazul în care calea spre anumite fişiere din timpul dezvoltării help-ului se schimbă pe parcursul căutărilor. De exemplu: `/proiectare/aplicatia/asistenta/prezentare/meniu.html` va fi memorată sub forma `prezentare/meniu.html`, dacă adăugăm în fişierul de configurare:

```
IndexRemove /proiectare/aplicatia/asistenta/; prezentare/meniu.html
```

Invers, `prezentare/meniu.html` devine `aplicatia/asistenta/prezentare/meniu.html`, dacă adăugăm în fişierul de configurare:

```
IndexPrepend aplicatia/asistenta/
```

- specificarea unei liste de fişiere ce se doresc a fi indexate. De exemplu, adăugând în fişierul de configurare liniile:

```
File meniu.html
File tabela.html
```

se vor indexa şi aceste fişiere.

- de asemenea, se poate specifica o listă de cuvinte exceptate. Cuvintele exceptate sunt acele cuvinte care nu se doresc indexate din cauza lipsei lor de importanță în procesul de căutare. Adăugând în fişierul de configurare:

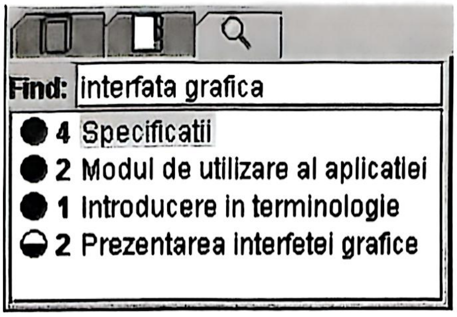
```
StopWords cuvintele un, o, toate, toti, sunt, si,
```

acestea se vor exclude de la indexare. Acelaşi efect l-am fi obținut dacă am fi scris în fişierul de configurare:

```
StopWordsFile cuvinte.txt
```

unde fişierul `cuvinte.txt` conține aceeaşi înşiruire de cuvinte de mai sus. Implicit, sunt exceptate o serie de cuvinte din limba engleză (*a, all, am...*).

Pentru a iniția căutarea, utilizatorul introduce o interogare în limbaj natural în navigatorul de căutare în zona de text având eticheta *Find*. Rezultatele sunt raportate înapoi utilizatorului în modul următor:

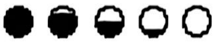


Find: interfata grafica

● 4 Specificatii
● 2 Modul de utilizare al aplicatiei
● 1 Introducere in terminologie
◒ 2 Prezentarea interfetei grafice

Rezultatele căutării

Discurile din prima coloană indică gradul de potrivire pentru subiectul căutat. Cu cât este mai plin discul, cu atât potrivirea este mai bună. Sunt patru grade de potrivire (de la cel mai înalt la cel mai mic):



Indicii pentru relevanța rezultatului

Numărul din coloana a doua indică de câte ori s-a regăsit potrivirea în tema dată. Textul reprezintă numele temei (aşa cum este specificat în fişierul conținut).

Motorul de căutare foloseşte o tehnică numită evaluare relaxată(engl. *relaxation ranking*) pentru a identifica şi clasifica paginile găsite ca răspunsuri la interogările utilizatorului. Astfel se încearcă găsirea de pasaje în temele help-ului în care, pe cât e posibil, termenii interogării utilizatorului se potrivesc în aceeaşi formă şi în aceeaşi ordine. Motorul de căutare evaluează automat constrângerile în identificarea paginilor în felul următor:

- nu toți termenii se potrivesc;
- termenii se potrivesc în forme diferite;
- termenii se potrivesc în ordine diferită;
- termenii se potrivesc având cuvinte despărțitoare.

Motorul de căutare asociază penalități corespunzătoare paginilor pentru deviații de la interogarea specificată de utilizator. Procesul de evaluare are loc atunci când interogările sunt complexe şi includ mai mulți termeni. De asemenea, motorul de căutare al sistemului JavaHelp foloseşte o tehnologie numită *morphing* pentru a găsi cuvinte cu rădăcini comune.

14.4.7. Compresia şi încapsularea

După crearea HelpSetului urmează un pas opţional, şi anume compresia şi încapsularea întregii documentaţii într-un singur fişier folosind formatul JAR pus la dispoziţie de platforma Java. Pentru aceasta trebuie să vă poziţionaţi în directorul rădăcină al sistemului help (în cazul structurii arborescente prezentate mai sus, pe directorul `/asistenta`) şi să apelaţi comanda:

```
jar cvf ajutor.jar *
```

pentru a crea arhiva dumneavoastră, care va avea numele `ajutor.jar` şi va păstra întreaga structură de directoare ale sistemului help creat. Pe parcursul procesului de arhivare veţi primi mesaje despre modul în care decurge acest proces. Spre exemplu, pentru fişierul `mapare.jhm` vom fi înştiinţaţi:

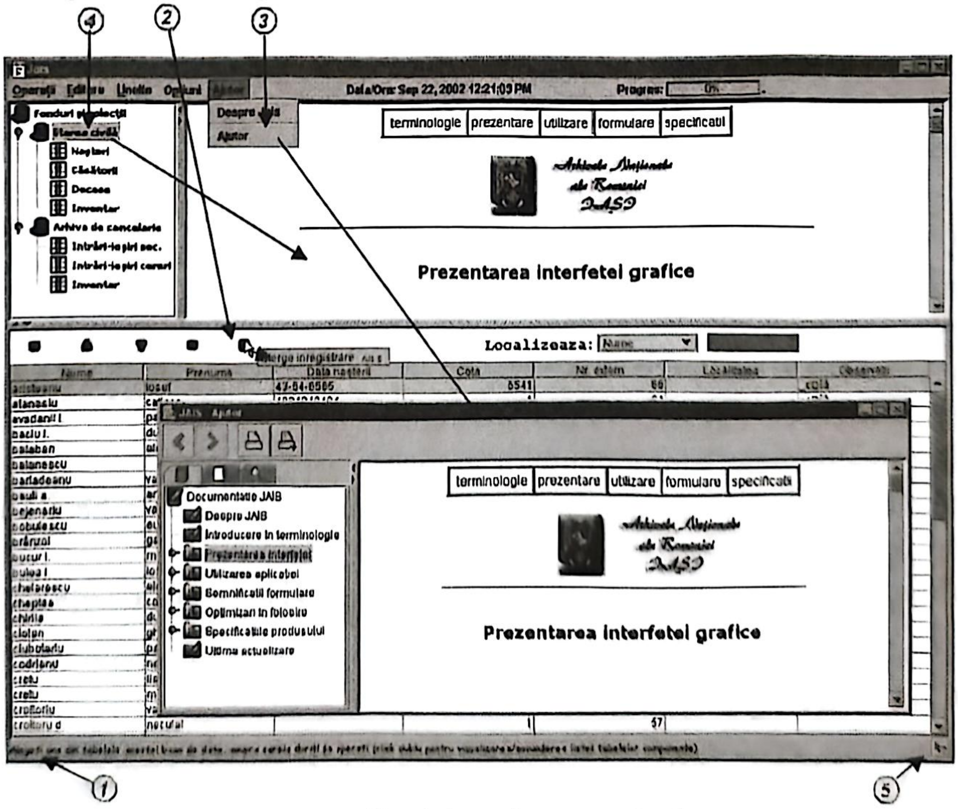
```
 adding: mapare.jhm (in=5757) (out=2216) (deflated 61%)
```

ceea ce înseamnă că fişierul s-a comprimat cu 61%.

Pentru a folosi fişierele din arhiva creată nu este nevoie să le extragem, ci le putem utiliza direct folosind tehnica JAR pusă la dispoziţie de Java 2 SDK.

14.5. Ataşarea help-ului aplicaţiilor

Prezentăm, în continuare, diverse modalităţi de a oferi asistenţă utilizatorului, ilustrate grafic sugestiv în imaginea care urmează :


④
②
③
①
⑤
Despre
Ajutor
terminologie
prezentare
utilizare
formulare
specificatii
Prezentarea interfetei grafice
Localizeaza:
Documentatie JAIB

Fereastra aplicaţiei cu elemente de ajutor

1. *Bara de stare* – reprezintă o etichetă (de obicei, un obiect `JLabel` căruia i se resetează mereu textul) afişată în partea de jos a ecranului, prin care se pot sugera utilizatorilor informaţii despre operaţiile pe care pot să le realizeze la un moment dat sau cum s-a desfăşurat operaţia curentă.
2. *Tooltips* – este modalitatea de a adăuga o etichetă explicativă unui buton sau, în general, unei componente, care va afişa un mesaj în momentul deplasării mouse-ului pe deasupra ei. Pentru a adăuga *tooltip*-uri se foloseşte clasa `JToolTip` sau, mai simplu, metoda `setToolTipText()` asociată fiecărei componente grafice.
3. *Fereastra help* – reprezintă o fereastră nemodală care va fi vizualizată la cererea utilizatorului prin alegerea unei opţiuni dintr-un meniu sau apăsarea unui buton care generează help, respectiv apăsarea tastei F1.
4. *Help scufundat în interfaţa grafică a aplicaţiei* – reprezintă combinarea sau refolosirea componentelor ferestrei sistemului help de la pasul anterior direct în interiorul interfeţei grafice a aplicaţiei.
5. *Afişare help la selectarea componentei dorite cu mouse-ul* – prin această metodă trebuie creat mai întâi un buton având următorul aspect: , prin apăsarea căruia se va modifica cursorul, luând forma ↖?, cu care mai apoi se selectează componenta al cărei help se intenţionează vizualizat.

Acest subcapitol dezbate ultimele trei metode de a adăuga help aplicaţiilor, care fac şi obiectul sistemului JavaHelp. A se consulta *javadoc*-ul care descrie API-ul JavaHelp.

Pentru a adăuga sistemul help aplicaţiei trebuie, mai întâi, importate clasele care oferă această facilitate:

```
import javax. help.*;
```

Iniţial, în `CLASSPATH` trebuie adăugată calea spre una dintre bibliotecile puse la dispoziţie (spre exemplu, `jh.jar`).

Vom discuta mai întâi despre clasele principale pe care sistemul JavaHelp le oferă programatorilor şi mai apoi modul în care acestea pot fi folosite pentru a rezolva unele aspecte privind adăugarea de help aplicaţiilor:

1. Clasa `HelpSet`.
2. Clasa `HelpBroker`.
3. Clasa `CSH`.
4. Help la nivel de fereastră (senzitiv la context).
5. Adăugarea de help componentelor grafice.
6. Butoane sau opţiuni de meniuri care determină vizualizarea help-ului.
7. Help prin selectarea cu mouse-ul.
8. Scufundarea (engl. *embedding*) help-ului în interfaţa grafică a aplicaţiei.

14.5.1. Clasa **`HelpSet`**

Legătura dintre aplicaţie şi documentaţia scrisă sub formă de HTML şi structurată folosind XML se realizează printr-o clasă care poartă denumirea `HelpSet`. Această clasă se află în pachetul `javax.help` şi este o înfăşurătoare pentru conţinutul help-ului. Constructorii acestei clase sunt de forma:

- `public HelpSet()` – creează un `HelpSet` vid; pentru a-i asocia un HelpSet în mod dinamic se foloseşte metoda `public void add(HelpSet hs);` pentru a şterge dinamic un HelpSet se recurge la metoda `public boolean remove (HelpSet hs)`.
- `public HelpSet(ClassLoader inc, URL helpset) throws HelpSetException` – încarcă un HelpSet descris printr-un fişier HelpSet care se află la adresa `helpset` folosind încărcătorul (engl. *loader*) de clase `inc` şi returnând o eroare, dacă nu se poate realiza procesarea fişierului. Dacă `inc` este `null`, atunci se va folosi încărcătorul implicit al maşinii virtuale. URL-ul unui HelpSet este returnat de o metodă statică a clasei `HelpSet` de genul : `public static URL findHelpSet(ClassLoader c, String nume)`, unde `c` reprezintă încărcătorul de clase (care poate fi `null`), iar `nume` reprezintă calea spre fişierul cu extensia `.hs`.

Exemplu:

```
try {
   URL hsURL = HelpSet.findHelpSet(null,
      "../asistenta/asistenta.hs");
   // hsURL va memora calea spre fisierul HelpSet
   HelpSet hs = new HelpSet(null, hsURL);
   // am creat un obiect HelpSet
} catch (Exception e) {
   // verificam corectitudinea procesarii
   System.err.println("Eroare la procesare: " + e);
   return;
}
```

14.5.2. Clasa **HelpBroker**

Obiectul `HelpBroker` este un agent care negociază şi gestionează prezentarea conţinutului help pentru aplicaţie. `HelpBroker`-ul oferă, de asemenea, metodele necesare pentru a implementa help senzitiv la context. Se poate implementa un sistem help fără a folosi clasa `HelpBroker`, ceea ce implică scrierea de cod care să-i suplinească lipsa. Se asociază unui obiect HelpSet folosind metoda `public HelpBroker createHelpBroker()`.

```
HelpBroker  hb = hs.createHelpBroker();
```

O metodă importantă a acestei clase este `public void initPresentation()`, care iniţializează prezentarea pentru a scurta din timpul de vizualizare. Este de preferat utilizarea ei într-un fir de execuţie separat la pornirea aplicaţiei, altfel, în momentul vizualizării se va pierde timpul şi cu iniţializarea.

Metode pentru gestionarea prezentării	Explicaţie
`public void` **`setDisplayed`** `(boolean val_adevar)`	Vizualizează fereastra de help şi aruncă `UnsupportedOperationException` în cazul în care operaţia nu se poate efectua.
`public void` **`setViewDisplayed`** `(boolean val_adevar)`	Setează dacă apare sau nu panoul de navigare în fereastra JavaHelp.
`public void` **`setSize`** `(Dimension dim)`	Setează dimensiunea ferestrei JavaHelp; aruncă `UnsupportedOperationException` dacă operaţia nu se poate efectua.
`public void` **`setLocation`** `(Point punct)`	Setează coordonatele la care se va afişa fereastra JaveHelp; la fel aruncă excepţia `UnsupportedOperationException`.
`public void` **`setCurrentView`** `(String nume)`	Setează navigatorul cu numele cu care este definit în fişierul HelpSet.
`public void` **`setCurrentID`**`(String id)`	Setează ca tema care va apărea în panoul de conţinut să fie cea cu identificatorul `id`; aruncă `BadIDException`, dacă identificatorul nu este definit în fişierul Map.

Metode pentru implementarea help-ului senzitiv la context	Explicaţie
`public void` **`enableHelpKey`** `(Component comp, String id, HelpSet hs)`	Setează identificatorul şi HelpSet-ul pentru o fereastră `JFrame` (în acest caz, `comp` reprezintă `rootPane`-ul ferestrei) sau pentru o fereastră Window în cazul implementărilor cu AWT (în acest caz `comp` este chiar fereastra). De asemenea, această metodă determină ca atunci când se apasă tasta Help (tasta `F1` implicit) să se realizeze prezentarea help-ului.
`public void` **`enableHelp`**`(Component comp, String id, HelpSet hs)`	Activează sistemul help pentru componenta `comp` prin adăugarea identificatorului de temă `id` şi HelpSet-ului `hs`.
`public void` **`enableHelp`** `(MenuItem menu, String id, HelpSet hs)`	Acelaşi lucru, dar pentru `MenuItem`-ul `menu`.
`public void` **`enableHelpOnButton`** `(Component comp, String id, HelpSet hs)`	Asociază identificatorul de temă şi HelpSet-ul pentru o componentă, precum şi un ascultător `ActionListener`, care se execută după activarea componentei şi care va vizualiza fereastra help cu tema dată de identificator (dacă `comp` nu e `AbstractButton` sau `Button`, sau derivată din ele, va arunca excepţia `IllegalArgumentException`).

`public void` **`enableHelpOnButton`** `(MenuItem menu, String id, HelpSet hs)`	Acelaşi lucru, dar pentru un `MenuItem`.

14.5.3. Clasa `CSH`

Este o clasă dedicată în totalitate implementării help-ului senzitiv la context. Prezentăm următoarele metode statice care permit ataşarea de identificatori şi de HelpSet-uri componentelor AWT sau Swing, determinând prin apăsarea tastei *Help* să se afişeze help-ul referitor la componenta respectivă:

Metode pentru implementarea help-ului senzitiv la context	Explicaţie
`public static void` **`setHelpIDString`**`(Component comp, String helpID)`	Asociază componentei grafice `comp` un identificator definit în fişierul Map. Atunci când se va apăsa `F1` şi componenta deţine focus-ul sau se apasă clic cu mouse-ul având cursorul ↖? pe componentă, se va afişa fereastra help având drept conţinut tema cu identificatorul `helpID`. Când componenta nu are asociată o temă, se va căuta ascendent având drept criteriu relaţia *a part of* până se va găsi o asignare.
`public static void` **`setHelpSet`** `(Component comp, HelpSet hs)`	Asociază HelpSet-ul `hs` componentei `comp`.

Aceleaşi metode sunt valabile pentru `MenuItem`.

Clasa `CSH` conţine trei clase interioare importante, care sunt, de fapt, `ActionListener`-i, ce pot fi asociaţi componentelor grafice:

- `public static class CSH.DisplayHelpFromSource` are constructorul `public CSH.DisplayHelpFromSource(HelpBroker hb)` şi preia identificatorul de temă corespunzător componentei grafice pe care o ascultă (în principiu, un buton) şi vizualizează fereastra help având tema corespunzătoare identificatorului drept conţinut. Spre exemplu, secvenţa următoare determină prin setarea opţiunii *Ajutor* vizualizarea temei asociate acestei opţiuni:

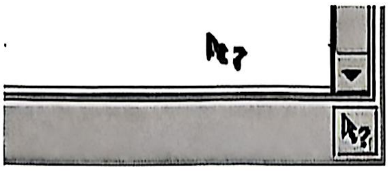
```
JMenuItem optiuneAjutor = new JMenuItem("Ajutor");
optiuneAjutor.addActionListener(
   new CSH.DisplayHelpFromSource(hb));
```

Ajutor
Despre Jais
Ajutor

Meniu de ajutor

- `public static class CSH.DisplayHelpAfterTracking` are constructorul `public CSH.DisplayHelpAfterTracking(HelpBroker hb)` şi oferă posibilitatea de a obţine help prin alegerea cu mouse-ul a componentei dorite. Prin asocierea acestui ascultător (engl. *listener*) unui buton `JButton` (sau derivat) sau `MenuItem`, automat prin apăsarea lui se obţine cursorul şi butonul rămâne apăsat până la alegerea componentei. După alegere se va afişa fereastra help corespunzătoare componentei grafice selectate şi se va aduce cursorul mouse-ului la starea iniţială.

```
contex.addActionListener(
   newCSH.DisplayHelpAfterTracking(hb));
```



Selectorul componentei pentru afişarea documentaţiei

- `public static class CSH.DisplayHelpFromFocus` are constructorul `public CSH.DisplayHelpFromFocus (HelpBroker hb)` şi este folosită pentru a determina vizualizarea help-ului (tema dată de identificatorul asociat, dacă există) pentru componente în momentul în care preiau focusul. Este folosit, în general, pentru componente diferite de `JFrame` sau `JWindow`.

14.5.4. Adăugarea help-ului ferestrelor

În exemplul următor, ultima linie ataşează help ferestrei de bază a aplicaţiei, determinând ca apăsarea tastei F1 să afişeze fereastra standard help, a cărei structură a fost descrisă la începutul acestui capitol:

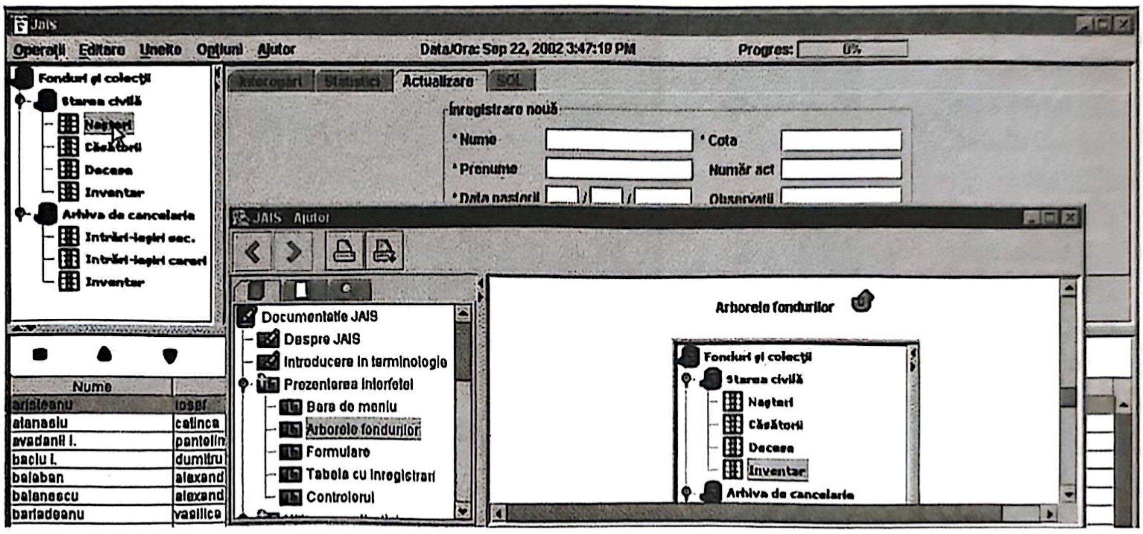
```
HelpBroker  hb = hs.createHelpBroker();
//obtinem obiectul HelpBroker din HelpSet-ul hs
//hb.setDisplayed(true); putem vizualiza help-ul oricand
//folosind brokerul
//in firul de executie urmator se initializeaza sistemul
//help
Thread fir = new Thread() {
  public void run() {
    try {
       hb.initPresentation();
    } catch (Exception e) {
       System.err.println("Eroare: " + e);
    }
  }
};
fir.start();
// ....
hb.enableHelpKey(this.getRootPane(), "prima.pagina", null);
//prin apsarea tastei F1 se obtine vizualizarea help-ului
```

14.5.5. Adăugarea help-ului componentelor

Dacă dorim ca de fiecare dată când apăsăm tasta F1 să obţinem informaţii referitoare la componenta grafică curentă, trebuie să asociem mai întâi componentelor identificatorii descrişi în fişierul Map. În exemplul următor adăugăm unui `JScrollPane`, care conţine un `JTree`, identificatorul `prezentare.arbore` descris iniţial în fişierul Map:

```
derulareArbore = new JScrollPane(arbore);
CSH.setHelpIDString(derulareArbore,"prezentare.arbore");
```

Dacă se apasă tasta F1, atunci când arborele este selectat va apărea fereastra de help, având drept conţinut tema `prezentare.arbore`. Chiar dacă arborele nu are asociat un identificator, se va afişa tema corespunzătoare `JScrollPane`-ului datorită căutării ascendente pe baza relaţiei *a part of* de care am pomenit mai sus:


JAIS
Operaţii
Editare
Unelte
Opţiuni
Ajutor
Progres:
0%
Fonduri şi colecţii
Starea civilă
Naşteri
Căsătorii
Decese
Inventar
Arhiva de cancelarie
Intrări-ieşiri sec.
Intrări-ieşiri cereri
Inventar
Interogări
Statistici
Actualizare
SQL
Înregistrare nouă
*Nume
*Prenume
*Data naşterii
*Cota
Număr act
Observaţii
JAIS Ajutor
Documentatie JAIS
Despre JAIS
Introducere in terminologie
Prezentarea interfetei
Bara de meniu
Arborele fondurilor
Formulare
Tabela cu inregistrari
Controlorul
Arborele fondurilor
Fonduri şi colecţii
Starea civilă
Naşteri
Căsătorii
Decese
Inventar
Arhiva de cancelarie

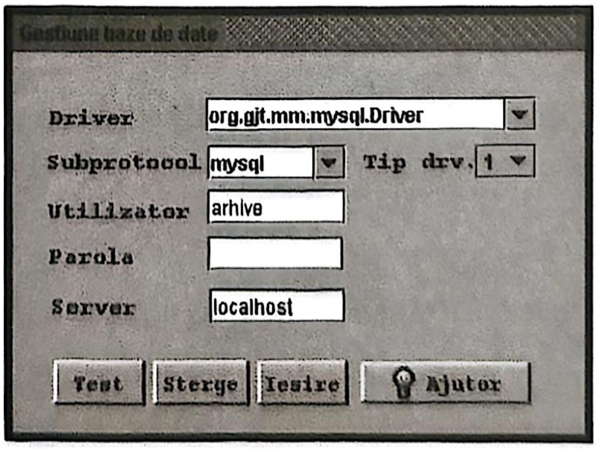
Data/Ora: Sep 22, 2002 3:47:19 PM

Nume	
aristeanu	iosif
atanasiu	catinca
avadanii i.	pantelin
baciu i.	dumitru
balaban	alexand
balanescu	alexand
barladeanu	vasilica

Obţinerea documentaţiei în funcţie de context

14.5.6. Activarea help-ului prin apăsarea butoanelor

Considerăm următoarea componentă derivată din `JDialog`:


Gestiune baze de date
Test
Sterge
Iesire
Ajutor

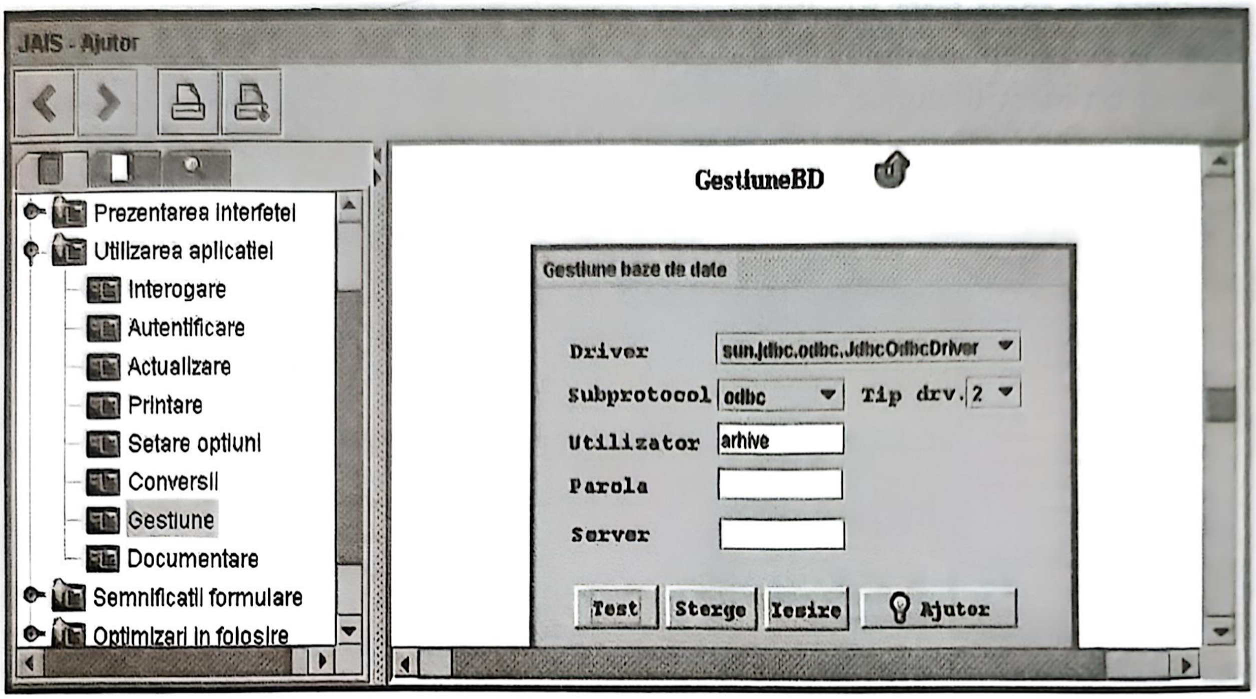
Driver org.gjt.mm.mysql.Driver
Subprotocol mysql
Tip drv. 1
Utilizator arhive
Parola
Server localhost

Adăugarea butonului de ajutor pentru ferestrele de dialog

Butonului *Ajutor* i se ataşează identificatorul unei teme prin secvenţa de cod:

```
JButton ajutor = new JButton();
ajutor.setIcon(new ImageIcon(DialogSQL.class.getResource(
       "ajutor.gif")));
ajutor.setText("Ajutor");
JaisFereastra.hb.enableHelpOnButton(ajutor,
       "folosire.gestiune", null);
```

Atunci când vom apăsa butonul *Ajutor*, se va vizualiza:


JAIS - Ajutor
Prezentarea interfetei
Utilizarea aplicatiei
Interogare
Autentificare
Actualizare
Printare
Setare optiuni
Conversii
Gestiune
Documentare
Semnificatii formulare
Optimizari in folosire
GestiuneBD
Gestiune baze de date
Test
Sterge
Iesire
Ajutor

Driver sun.jdbc.odbc.JdbcOdbcDriver
Subprotocol odbc
Tip drv. 2
Utilizator arhive
Parola
Server

Obţinerea documentaţiei pentru o anumită fereastră

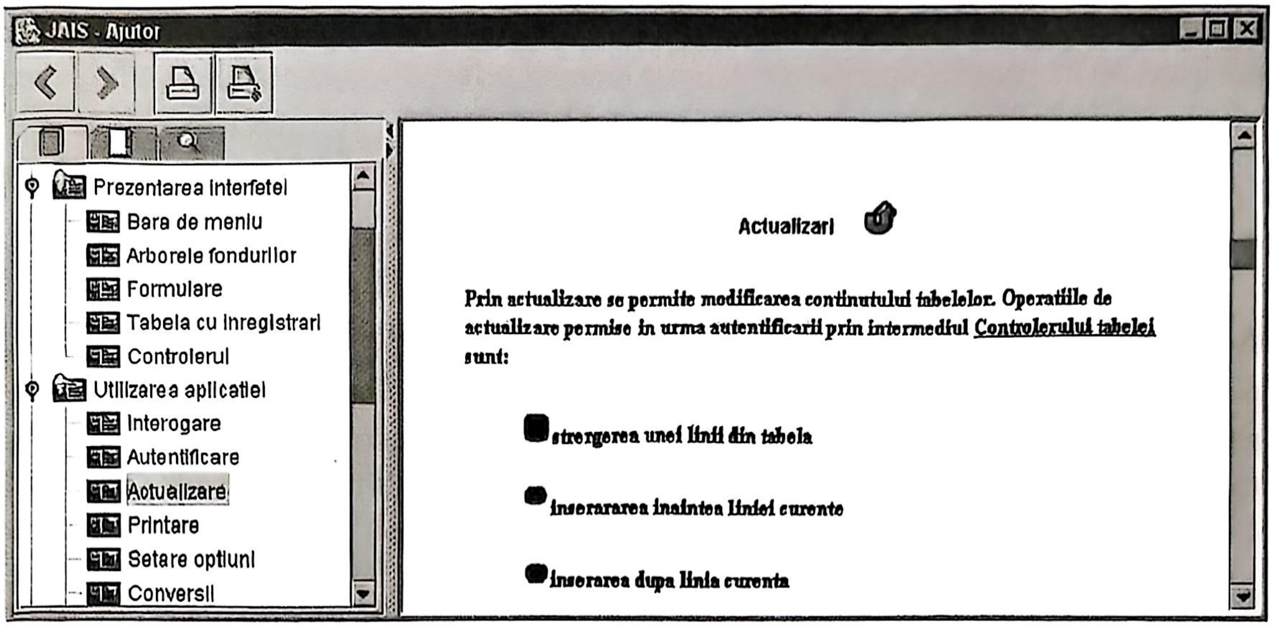
În acelaşi mod se procedează şi cu `JMenuItem`.

14.5.7. Help activat folosind mouse-ul

Pentru a prezenta help prin poziţionarea cu mouse-ul este suficientă secvenţa de cod următoare:

```
//cream un buton
JButton contex = new JButton(new ImageIcon("context.gif"));
//asociem ascultator butonului
contex.addActionListener(
   new CSH.DisplayHelpAfterTracking(hb));
//adaugam ToolTip butonului
contex.setToolTipText("Apasati si alegeti componenta dorita");
```

Prin apăsarea butonului mouse-ului, după schimbarea cursorului în poziţia curentă, se va vizualiza fereastra de help având următorul conţinut (de menţionat că tabela are asociat identificatorul `folosire.actualizare`):


JAIS - Ajutor
Prezentarea interfetei
Bara de meniu
Arborele fondurilor
Formulare
Tabela cu inregistrari
Controlerul
Utilizarea aplicatiei
Interogare
Autentificare
Actualizare
Printare
Setare optiuni
Conversii

Actualizari

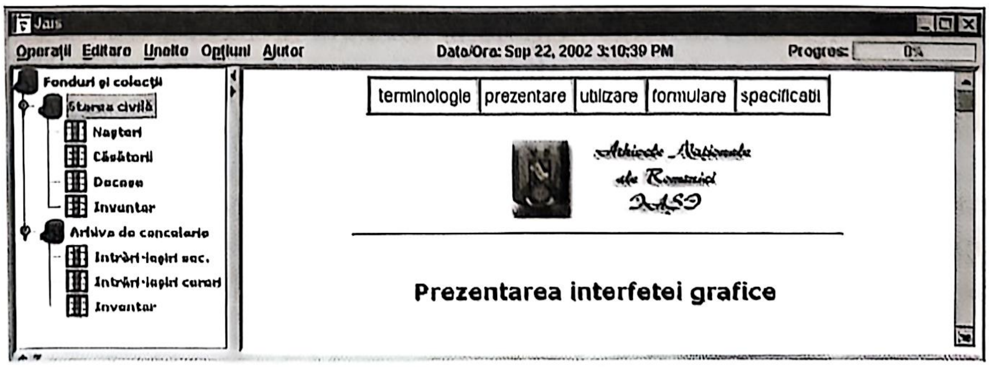
Prin actualizare se permite modificarea continutului tabelelor. Operatiile de actualizare permise in urma autentificarii prin intermediul Controlerului tabelei sunt:

- **stergerea unei linii din tabela**
- **inserararea inaintea liniei curente**
- **inserarea dupa linia curenta**

14.5.8. Scufundarea help-ului în aplicaţie

Putem scufunda porţiuni din sistemul help direct în interfaţa grafică a aplicaţiei noastre. În exemplul care urmează adăugăm interfeţei noastre panoul de conţinut al sistemului help, căruia îi setăm şi o anumită temă:

```
JHelpContentViewer aj = new JHelpContentViewer(hs);
//aj reprezinta continutul HelpSetului hs si este
//un obiect JComponent din faptul ca JHelpContentViewer
//mosteneste clasa JComponent ceea ce ne dreptul
//de a-l adauga  obiectelor JContainer sau derivate
JPanel pan = new JPanel(); //cream un panou nou
try {
   aj.setCurrentID("prezentare.introducere");
   // am setat tema dorita
} catch (Exception en) {
   System.err.println("eroare");
}
pan.add(aj); // adaugam continutul intr-un panou
// adaugam continutul in partea dreapta a JSplitPane-ului
sus.setRightComponent(aj);
```


Jais
Operaţii
Editare
Unelte
Opţiuni
Ajutor
Data/Ora: Sep 22, 2002 3:10:39 PM
Progres:
0%
Fonduri şi colecţii
Starea civilă
Naşteri
Căsătorii
Decese
Inventar
Arhiva de cancelarie
Intrări-ieşiri sec.
Intrări-ieşiri curen
Inventar
terminologie
prezentare
utilizare
formulare
specificatii
Arhivele Naţionale ale României Iaşi

Prezentarea interfetei grafice

Scufundarea documentaţiei în aplicaţie

Se pot adăuga aplicaţiei şi alte componente ale sistemului help, cum ar fi **panoul de navigare**. Evităm astfel apariţia pe ecran a unei alte ferestre dedicate **help-ului**, care ne-ar putea stânjeni.

14.6. Concluzii

Pentru început s-au prezentat noţiuni introductive legate de sistemul de asistenţă pentru aplicaţii, urmate de o descriere a pachetului JavaHelp. Ca observaţie, acest pachet nu a fost introdus încă în nici o distribuţie a platformei Java, din pricina modificărilor continue prin care trece. Este prezentată în continuare alcătuirea standard a interfeţei sistemului JavaHelp şi a principiilor pe care acest sistem le respectă.

Într-o manieră progresivă, sunt prezentaţi paşii necesari pentru a crea sistemul de asistenţă al unei aplicaţii. S-a folosit drept fundal un exemplu efectiv, uşurând astfel înţelegerea conceptelor implicate în acest proces. Pentru fiecare document XML folosit s-a prezentat structura sa.

Ultima secţiune a capitolului descrie metodele efective care pot fi utilizate pentru a adăuga help aplicaţiilor. Sunt enumerate clasele la care recurge acest proces şi exemple efective de folosire a lor.

14.7. Test-grilă

Întrebarea 14.7.1. Care este funcţionalitatea sistemului JavaHelp?

a) Oferă posibilitatea creării *javadoc*-ului pentru aplicaţiile Java.
b) Oferă posibilitatea de a adăuga ajutor contextual aplicaţiilor.
c) Reprezintă documentaţia oficială a limbajului Java.

Întrebarea 14.7.2. Ce pachete trebuie incluse în `CLASSPATH` pentru a putea crea sistemul Help pentru aplicaţiile Java?

a) Numai `jsearch.jar`.
b) Pachetele `jhall.jar`, `jhbasic.jar` şi `jh.jar`.
c) Pachetul `jh.jar` este de ajuns.

Întrebarea 14.7.3. Ce reprezintă scufundarea ferestrelor help în aplicaţie?

a) Ascunderea ferestrei de bază help după fereastra de bază a aplicaţiei.
b) Integrarea help-ului direct în aplicaţie.
c) Renunţarea la sistemul help pentru aplicaţie.

Întrebarea 14.7.4. Care este succesiunea corectă de paşi necesari în realizarea sistemului help pentru o aplicaţie?

a) Ataşarea de help aplicaţiei, crearea temelor HTML, scrierea descriptorilor XML.
b) Crearea temelor HTML, scrierea descriptorilor XML, ataşarea de help aplicaţiei.
c) Scrierea descriptorilor XML, crearea temelor HTML, ataşarea de help aplicaţiei.

Întrebarea 14.7.5. Care dintre următoarele afirmaţii sunt adevărate?
a) Fişierul Map este folosit pentru a asocia identificatori temelor.
b) Fişierul cuprins (*Table Of Contents* – TOC) descrie conţinutul documentaţiei.
c) Fişierul index conţine indexul temelor HTML.

Întrebarea 14.7.6. Care este rolul utilitarului `jhindexer`?
a) Permite crearea fişierului index.
b) Permite regăsirea de intrări în fişierul index.
c) Afişează rezultatele unei căutări în fişierul index.

Întrebarea 14.7.7. Fie secvenţa de cod:

```
JButton c = new JButton(new ImageIcon("context.gif"));
c.addActionListener(new CSH.DisplayHelpAfterTracking(hb));
contex.setToolTipText("Apasati si alegeti componenta dorita");
```

Care dintre afirmaţiile următoare este adevărată?
a) Apăsarea butonului determină terminarea aplicaţiei.
b) Apăsarea butonului determină schimbarea formei cursorului.
c) După apăsarea butonului, la prima apăsare a butonului stâng al mouse-ului se va afişa help contextual.

14.8. Exerciţii propuse spre implementare

Exerciţiul 14.8.1. Prezentaţi acest capitol sub forma unui sistem JavaHelp.

Exerciţiul 14.8.2. Adăugaţi ajutor contextual aplicaţiilor dumneavoastră, urmând paşii descrişi pe parcursul acestui capitol.

Exerciţiul 14.8.3. Personalizaţi după preferinţă interfaţa standard a sistemului JavaHelp folosind informaţiile din ghidul de utilizare.

14.9. Proiecte propuse spre implementare

Proiectul 14.9.1. Realizaţi o aplicaţie având interfaţă grafică şi care să permită crearea într-un mod facil a sistemului help pentru o aplicaţie Java. Pentru implementare se va folosi biblioteca Swing.

Proiectul 14.9.2. Creaţi o aplicaţie care să permită asocierea de help aplicaţiilor într-o manieră comodă. Se va permite încărcarea şi parcurgerea surselor pentru adăugarea de help contextual.

Proiectul 14.9.3. Creaţi o aplicaţie care să permită prezentarea documentaţiei *javadoc* într-o manieră asemănătoare JavaHelp-ului.

Proiectul 14.9.4. Realizaţi un applet care să ofere help contextual aplicaţiilor preluând aceste informaţii prin reţea.

15. Internaţionalizarea aplicaţiilor

15.1. Cuvinte-cheie

- internaţionalizare
- localizare
- seturi de caractere
- formatări

15.2. Introducere

Internaţionalizarea reprezintă procesul de proiectare a aplicaţiilor, astfel încât acestea se pot adapta la diverse limbi şi regiuni, fără a fi nevoie de modificări în aplicaţie după ce ciclul de dezvoltare a acesteia s-a terminat. Un bun exemplu în acest sens este cazul unui applet care detectează localizarea maşinii pe care se execută şi prezintă informaţiile celor care îl accesează în limba preferată a acestora. Un alt exemplu ar putea fi aplicaţiile care oferă meniuri şi, în general, interfaţa realizată în mai multe limbi, utilizatorul selectând limba dorită fie la instalarea aplicaţiei, fie pe parcursul utilizării ei. La fel, ne putem gândi la sistemul help ataşat unei aplicaţii care este de preferat să fie oferit în mai multe limbi, înţelegerea lui fiind esenţială utilizatorilor aplicaţiei. Deseori termenul „internaţionalizare" este abreviat prin *i18n*, deoarece sunt 18 caractere între prima literă „i" şi ultima literă „n" din cuvântul echivalent din engleză (*internationalization*).

În acest context, apare noţiunea de *localizare* care se constituie tocmai în procesul de structurare a datelor pe diverse limbi şi regiuni, ceea ce reprezintă un pas important în adaptarea unui program pentru diverse locaţii. Prin localizare (în sens de locaţie) vom înţelege o regiune geografică sau politică în care se vorbeşte aceeaşi limbă. Procesul de localizare a unei aplicaţii presupune traducerea textelor etichetelor, mesajelor de eroare şi a help-ului, respectiv formatarea datelor corespunzătoare respectivelor locaţii, precum ar fi valorile monetare, datele calendaristice, numerele. Termenul „localizare" mai este abreviat prin *l10n*, deoarece sunt 10 litere între „l" şi „n" în echivalentul din engleză (*localization*).

Din experienţă, se poate observa că este de preferat ca o aplicaţie să fie construită de la început respectând principiile internaţionalizării, deoarece procesul de transformare a acesteia pentru a fi în acord cu internaţionalizarea este complicat şi înseamnă rescriere de cod.

Un program internaţionalizat are următoarele caracteristici:

- elementele text, cum ar fi mesajele din bara de stare sau conţinutul etichetelor, nu sunt încapsulate în aplicaţie, ci sunt ţinute în afara ei, de unde sunt încărcate în mod dinamic;
- suportul pentru noi limbi nu implică recompilarea;
- informaţiile „culturale" precum data, sistemul de măsuri apar în conformitate cu regiunea în care se află utilizatorul şi cu limba pe care acesta o vorbeşte.

Din lista de cerinţe prezentată mai sus se poate observa o încercare de a separa datele sensibile (cele care se doresc prezentate conform unei anumite localizări) de codul aplicaţiei, pentru a evita recompilările acesteia. De asemenea, se remarcă necesitatea unei metode prin care să se realizeze descrierea diverselor localizări. Pachetul `java.util` oferă suportul pentru aceste două aspecte prin intermediul clasei `ResourceBundle`, care stă la baza separării datelor de aplicaţie, respectiv clasa `Locale`, care oferă un mod standard de stabilire a localizărilor. Descrierea modului în care trebuie folosite clasele `Locale` şi `ResourceBundle` este prezentată în secţiunile acestui capitol. Mai întâi, vom discuta însă despre modul în care limbajul Java foloseşte seturile de caractere.

15.3. Codificări

O mulţime de codificări de caractere (engl. *coded character set*) reprezintă o asociere (engl. *mapping*) între o mulţime abstractă de caractere şi o mulţime de numere întregi. US-ASCII, ISO 8859-1, Unicode (ISO 10646-1) sunt exemple de mulţimi de codificări de caractere.

O schemă de codificare a caracterelor (engl. *character-encoding scheme*) reprezintă o asociere între o mulţime de codificări de caractere şi o mulţime de secvenţe de octeţi. UTF-8 (*Universal Transformation Format*) - folosit de protocoalele de reţea şi de sistemul de fişiere Unix, UCS-2, UTF-16, EUC sunt exemple de astfel de scheme. În general, o schemă este asociată unei singure mulţimi de codificări, purtând numele acesteia, aşa cum este cazul US-ASCII, care reprezintă şi numele setului de caractere şi al schemei de codare. Un exemplu în care schema şi mulţimea de caractere nu au acelaşi nume ar putea fi cazul schemei UTF-8, care este folosită pentru a codifica Unicode. Alte scheme sunt asociate cu mai multe mulţimi de caractere. Astfel, schema EUC poate fi folosită pentru a codifica diverse mulţimi de caractere din zona asiatică. Schemele de codificare în notaţia Microsoft poartă numele de *code page*, numele lor începând cu prefixul *cp*, spre exemplu Cp1250 pentru Europa de Est.

Vom folosi în general noţiunea de *set de caractere* (engl. *character set*) atât pentru mulţimea de codări, cât şi pentru schemă, cititorul desprinzând din context despre care dintre acestea este vorba. În Java, numele seturilor de caractere trebuie să înceapă cu o literă sau o cifră şi nu contează dacă sunt scrise cu litere mari sau mici. Numele seturilor de caractere respectă convenţiile specificate prin RFC 2276. Setul de caractere nativ pentru limbajul Java este Unicode, această alegere rezolvând multe dintre problemele legate de manipularea şi reprezentarea caracterelor. Intern, caracterele sunt reprezentate folosindu-se schema de codare UTF-16. De asemenea, limbajul Java suportă conversii între Unicode şi alte seturi de caractere. Aceste conversii se realizează prin intermediul claselor `java.io.InputStreamReader`, `java.io.OutputStreamWriter` şi `java.lang.String` şi sunt necesare pentru interacţiunea aplicaţiilor Java cu bazele de date, cu alte programe etc. care folosesc alte seturi de caractere. În cazul interacţiunii cu astfel de aplicaţii, se vor face conversiile datelor spre Unicode, vor fi prelucrate şi mai apoi transformate din nou spre setul de caractere original.

15.3.1. Despre Unicode

În limbajul de programare Java variabilele, având tipul `char`, reprezintă caractere Unicode. Unicode este un standard de codare a caracterelor pe 16 biţi (doi octeţi) care oferă suport pentru cele mai multe dintre limbile existente, permiţând codarea a 2^{16} caractere. Mai vechiul standard ASCII (engl. *American Standard Code for Information Interchange*) foloseşte codarea pe 7 biţi, iar *ISO Latin-1* utilizează codarea pe 8 biţi. Chiar dacă sursele programelor sunt scrise de cele mai multe ori folosind aceste ultime două codări, înainte de a se trece la procesarea lor în JVM, acestea sunt translatate în

Unicode. Specificaţia Unicode o puteţi găsi la adresa *http://www.unicode.org/*, ea reprezentând documentul de referinţă pentru cei ce folosesc acest mod de codare a caracterelor.

Nu vom insista prea mult asupra setului de caractere Unicode, pentru că, în general, nu este nevoie de acesta pentru a lucra cu limbajul Java, codarea fiind folosită intern, iar trecerile spre alte codări făcându-se prin clase specializate.

Unicode permite trei tipuri de codări: UTF-8, UTF-16, UTF-32. Numărul de biţi din denumiri reprezintă numărul de biţi din care este alcătuită unitatea cu care se face codarea. Spre exemplu, pentru UTF-8 caracterele se pot coda variabil prin unu, doi, trei sau patru octeţi. În cazul UTF-16, caracterele se pot coda prin unul sau două cuvinte (engl. *word* = 2 octeţi). În cazul UTF-32, codarea se face fix prin secvenţe de 32 biţi.

Tabelele cu seturile de caractere Unicode se găsesc la adresa *http://www.unicode.org/charts/*.

15.3.2. Seturi de caractere

Compilatorul Java şi alte instrumente Java pot procesa numai fişiere conţinând codarea Latin-1, eventual combinată cu notaţia Unicode (secvenţe `\udddd`) pentru caractere. Pentru a transforma fişierele dintr-o codare oarecare în Latin-1 se poate folosi un utilitar numit `native2ascii`, care se află în directorul `bin` al distribuţiei Java. Sintaxa acestei comenzi este:

```
native2ascii [optiuni] [fisier intrare [fisier iesire]]
```

Dacă fişierul de ieşire este omis, se va trimite către ieşirea standard (ecran). Dacă fişierul de intrare lipseşte, se va citi de la intrarea standard (tastatură). Opţiunile pot fi:

- `reverse` realizează operaţiunea inversă transformând fişiere scrise folosind setul Latin-1 cu secvenţe în notaţie Unicode spre o altă codare;
- `encoding nume_codare` specifică numele codării folosită în procesul de conversie, iar numele este dat de tabela cu codări acceptate pe care o găsiţi în *javadoc*: `docs\guide\intl\encoding.doc.html`. Dacă nu se specifică nici o codare, se va lua în calcul codarea implicită.

Spre exemplu, putem să salvăm ca text simplu (engl. *plain*), cu numele `unicode.txt`, un document redactat în Word folosind diacriticele (ultimele versiuni de Word permit selectarea mai multor codări în momentul salvării). Presupunem că îl salvăm în Unicode (UTF-8). Pentru a-l converti la Latin-1 vom folosi comanda:

```
native2ascii -encoding UTF8 unicode.txt latin.txt
```

În Java, orice set de caractere are asociat un nume canonic care poate fi diferit de numele din specificaţii şi o mulţime de aliasuri, iar gestionarea lor se face prin intermediul clasei `java.nio.charset.Charset`. Pachetul `java.nio` (denumirea vine de la New I/O) reprezintă noua bibliotecă pentru lucrul cu intrări-ieşiri pentru o aplicaţie Java. Acesta mai aduce o clasă nouă, şi anume `java.nio.charset.spi`.

`CharsetProvider`, care poate fi folosită pentru a adăuga noi seturi de caractere folosind extensii. Prezentăm, în continuare, seturile de caractere de bază care sunt suportate prin convenţie de orice platformă Java:

Nume canonic	Descriere
`US-ASCII`	Şapte biţi ASCII (ISO646-US), numit şi blocul de bază Latin al setului de caractere Unicode.
`Cp 1252`	Windows Latin-1.
`ISO-8859-1`	ISO Latin Alphabet No. 1 (ISO-LATIN-1)
`ISO-8859-15`	ISO Latin Alphabet No. 9.
`UTF-8`	UCS Transformation Format pe 8 biţi.
`UTF-16BE`	UCS Transformation Format pe 16 biţi, ordinea big-endian.
`UTF-16LE`	UCS Transformation Format pe 16 biţi, ordinea little-endian.
`UTF-16`	UCS Transformation Format pe 16 biţi.

Java suportă şi multe alte seturi de caractere (în general, specifice anumitor regiuni), în mod diferit pentru pachetul `java.io` şi `java.lang`. Prin „suportă" înţelegem că permite realizarea de conversii pentru că intern, după cum spuneam, întotdeauna se lucrează cu Unicode.

Pentru a determina seturile de caractere suportate de maşina virtuală, se poate folosi metoda statică `availableCharsets()` din clasa `java.nio.charset.Charset`.

Exemplul 15.3.1. Programul de mai jos afişează seturile de caractere suportate.

```
import java.nio.charset.Charset;
import java.util.Iterator;

public class SeturiCaractere {
  public static void main(String[] args) {
    Iterator<String> $it =
      Charset.availableCharsets().keySet().iterator();
    while ($it.hasNext()) {
      System.out.println($it.next());
    }
  }
}
```

Se va obţine o listă impresionantă de seturi de caractere suportate.

Întotdeauna există o codare implicită care este selectată automat de mediul de lucru Java, în funcţie de sistemul de operare peste care este instalată platforma şi de localizarea setată pentru acesta. Spre exemplu, în cazul în care avem setată localizarea Romanian pentru Windows, implicit se va considera setul de caractere *Cp1250*, care reprezintă setul de caractere pentru estul Europei. Codarea implicită este importantă, deoarece face legătura între mediul de execuţie Java, în care caracterele sunt reprezentate prin Unicode, şi sistemul de operare, care foloseşte pentru sistemul de fişiere o altă codare (care va deveni codarea implicită). Codarea implicită este păstrată de proprietatea `file.encoding` din `System` şi se obţine printr-un apel de forma:

```
System.out.println("Codarea implicita: " +
   System.getProperty("file.encoding"));
```

Suportul pentru transformări între diverse codări se realizează prin intermediul claselor `java.io.PrintStream`, `java.io.InputStreamReader`, respectiv `java.io.OutputStreamWriter`.

`PrintStream` reprezintă fluxul pentru ieşire care oferă diferite suprascrieri convenabile ale metodei `print()`, folosit în mod uzual pentru scrierea către ieşirea standard. Acesta are un constructor care permite folosirea altor seturi de caractere diferite de setul implicit de caractere. Prezentăm modul în care se poate seta ca spre ieşirea standard să se scrie cu caractere est-europene (dacă localizarea este Romanian, deja această codare este folosită implicit):

```
PrintStream ps1 =
   new PrintStream(System.out , true, "Cp1250");
System.setOut(ps1);
```

În exemplul care urmează prezentăm redirectarea ieşirii standard spre un fişier având numele `iesire.txt`. De asemenea, scrierea se va face folosind setul de caractere pentru Europa de Est.

```
PrintStream ps =
   new PrintStream(new FileOutputStream (
     new File("iesire.txt")), true, "Cp1250");
System.setOut(ps);
```

Pentru a vizualiza rezultatul scris în fişier trebuie folosit un editor care vizualizează caracterele est-europene (poate folosi codarea Cp1250). Altfel (pentru că editoarele simple folosesc în general ASCII), „ş" şi „ţ" nu se vor vizualiza corect.

Clasa `InputStreamReader` reprezintă o punte între fluxuri de biţi şi fluxuri de caractere. Poate fi folosită şi pentru a transforma textul non-Unicode considerat a aparţine unui set de caractere şi reprezentat prin fluxul de biţi în text Unicode. Dacă nu este specificat explicit prin convenţie, se consideră setul de caractere implicit. Un exemplu de transformare a unui text ASCII în text Unicode:

```
FileInputStream fisier =
   new FileInputStream("intrare.txt");
InputStreamReader uni =
   new InputStreamReader(fisier, "US-ASCII");
String codare = uni.getEncoding();
```

Metoda `getEncoding()` specifică codarea folosită de flux.

Pentru trecerea din Unicode în alte seturi de caractere se poate folosi clasa `OutputStreamWriter`. Exemplul următor permite scrierea unui şir de caractere într-un fişier folosind codarea UTF-8:

```
try {
    FileOutputStream fisier =
        new FileOutputStream("text.txt");
    Writer iesire =
        new OutputStreamWriter(fisier, "UTF-8");
    iesire.write("salut");
    iesire.close();
} catch (IOException e) {
    e.printStackTrace();
}
```

15.4. Setarea localizării

Un program internaţionalizat poate vizualiza informaţii diferite în funcţie de locaţia din care este apelat. Spre exemplu, dacă este vorba de un applet, acesta va prezenta informaţii în limba română, dacă este rulat pe o maşină din România, şi în limba franceză, dacă este apelat pe o maşină din Franţa. În limbajul Java, localizarea clienţilor aplicaţiilor se încapsulează într-un obiect `Locale` din pachetul `java.util`. Obiectele `Locale` sunt simpli identificatori pentru diverse combinaţii de limbi şi regiuni, neconţinând atribute caracteristice localizării. Unei aplicaţii Java nu i se atribuie un singur obiect `Locale`, ci clasele se pot conforma unei instanţe `Locale`, în acest caz numindu-se sensibile la localizare (engl. *local-sensitive*). Din acest motiv putem avea într-o aplicaţie clase conform diferitelor instanţe `Locale` şi deci informaţiile pot fi prezentate în mai multe limbi concomitent. Dacă nu se specifică o instanţă `Locale` pentru obiectele sensibile la localizare, atunci implicit aceasta este stabilită de JVM în momentul pornirii ca fiind obiectul `Locale` corespunzător platformei gazdă. O referinţă spre acesta puteţi obţine printr-un apel `Locale.getDefault()`. De asemenea, se poate seta obiectul `Locale` implicit printrun apel `Locale.setDefault(Locale l)`, dar trebuie să avem grijă, pentru că el va fi adoptat implicit de toate instanţele claselor sensibile la localizare ale aplicaţiei.

Modul de creare a obiectelor `Locale` este extrem de simplu. Spre exemplu, pentru a crea un obiect `Locale` corespunzând limbii franceze şi ţării Canada vom folosi un constructor de forma `Locale loc = new Locale("fr", "CA")`, iar pentru limba română şi România, un apel `Locale loc = new Locale("ro", "RO")`. Primul şir reprezintă o pereche de litere mici corespunzând limbii conform standardului ISO-639 a cărui specificaţie o găsiţi la adresa *http://ftp.ics.uci.edu/pub/ietf/http/related/iso639.txt.*

Cel de al doilea şir format dintr-o pereche de litere majuscule reprezintă codul ţării şi respectă standardul ISO-3166 a cărui specificaţie o puteţi găsi la adresa: *http://www.iso.org/iso/en/prods-services/iso3166ma/02iso-3166-code-lists/index.html.*

Clasa `Locale` mai pune la dispoziţie un constructor prin care se permite introducerea unui al treilea parametru care poartă numele de cod variabil. De obicei, acesta este folosit pentru a specifica referinţe la platforma pe care rulează aplicaţia. Spre exemplu, putem folosi un apel `Locale l = new Locale("ro","RO","UNIX")`. Acest parametru este opţional şi, dacă apare, obiectul care îl foloseşte va şti cum să-l trateze. Un obiect `Locale` poate fi creat doar pentru o anumită limbă, în acest caz folosind la construcţia lui un apel în care al doilea parametru este un şir vid ca, de exemplu, `new Locale("ro", "")`. Clasa `Locale` pune la dispoziţia utilizatorilor o serie de constante corespunzând anumitor limbi şi ţări, cum ar fi `Locale.FRANCE`, `Locale.ITALY` etc.

Metoda	Descriere
`Public String` **`getCountry`**`()`	Codul ISO al ţării.
`Public String` **`getLanguage`**`()`	Codul ISO al limbii.
`public final String` **`getDisplayLanguage`**`()` `public String` **`getDisplayLanguage`** `(Locale loc)`	Returnează un şir reprezentând numele limbii corespunzătoare localizării curente. Dacă se precizează ca parametru o instanţă `Locale`, şirul returnat va fi numele limbii localizării curente în localizarea `loc`.
`public final String` **`getDisplayCountry`**`()` `public String` **`getDisplayCountry`**`(Locale inLocale)`	Returnează un şir reprezentând numele ţării corespunzătoare localizării curente. Dacă se precizează ca parametru o instanţă `Locale`, şirul returnat va fi numele ţării localizării curente în localizarea `loc`.

Odată creat un obiect `Locale`, acesta poate oferi informaţii via metodele prezentate în tabelul anterior.

Exemplul 15.4.1. Pentru a ilustra modul de lucru cu localizările, prezentăm următoarea secvenţă de cod:

```
import static java.lang.System.*;
import java.util.Locale;

public class TestLocalizare {

  public static void main(String[] args) {
    Locale loc = Locale.getDefault();
    out.println("Localizare implicita: " + loc);
    out.println(loc.getLanguage());
    out.println(loc.getCountry());
    out.println(loc.getDisplayLanguage());
    out.println(loc.getDisplayCountry());
    out.println(loc.getDisplayLanguage(Locale.FRENCH));
    out.println(loc.getDisplayCountry(Locale.FRENCH));
  }
}
```

Dacă localizarea implicită este `ro_RO`, atunci va fi afişat următorul mesaj:

```
Localizare implicita: ro_RO
ro
RO
română
România
romain
Roumanie
```

Localizarea implicită poate fi schimbată fie prin program, fie prin setarea unei localizări pentru sistemul de operare. De exemplu, pentru Windows, acest lucru se poate realiza din *Control Panel*, *Regional Settings*. Dacă vom seta, spre exemplu, aici localizarea *English (United States)*, în urma execuţiei secvenţei de mai sus va fi afişat următorul conţinut:

```
Localizare implicita: en_US
en
US
English
United States
anglais
Etats-Unis
```

Pentru a vedea care sunt localizările suportate de clasele sensibile la localizare, acestea pun la dispoziţia programatorilor metoda `getAvailableLocales()`, care returnează o listă a localizărilor. De exemplu, printr-un apel `NumberFormat.getAvailable Locale()` vom obţine o listă a localizărilor pe care această clasă le suportă. De asemenea, clasa `Locale` oferă posibilitatea determinării tuturor localizărilor instalate.

Exemplul 15.4.2. Programul de mai jos afişează lista tuturor localizărilor acceptate.

```
import java.util.Locale;

public class LocalizariAcceptate {
   static public void main(String[] argumente) {
      Locale lista[] = Locale.getAvailableLocales();
      for (int i = 0; i < lista.length; i++)
         System.out.println(lista[i].toString());
   }
}
```

Va rezulta o înşiruire de reprezentări ale localizărilor de forma: `ar_EG`, `be_BY`, `bg_BG`, `ca_ES`, ..., `ro_RO` etc. Toate localizările sunt suportate în acelaşi fel şi independent de sistemul de operare peste care este instalată maşina Java şi de localizările acestuia. Singura implicare a sistemului de operare este în selectarea localizării implicite şi a datei şi orei implicite. Java suportă un mare număr de localizări, aşa cum se poate citi şi în pagina documentaţiei *javadoc* `docs\guide\intl\locale.doc.html`.

15.5. Separarea datelor

Datele sensibile la localizare vor fi păstrate separat de aplicaţie, grupate (engl. *bundle*) în funcţie de diferitele localizări. Spre exemplu, textul vizualizat de interfaţa grafică intră în categoria datelor sensibile la localizare. Un buton va afişa eticheta *OK* pentru limba engleză, *De acord* pentru limba română şi o altă etichetă pentru o altă limbă. Această afişare diferenţiată nu se face printr-o tratare în codul aplicaţiei, ci prin simpla alegere a instanţei `Locale` pentru obiectul `JButton`. Gruparea datelor pe diferite localizări se realizează prin intermediul clasei `ResourceBundle`.

Conceptual, `ResourceBundle` reprezintă un set de subclase având drept nume o aceeaşi rădăcină urmată de numele unei localizări în formatul obişnuit (`codLimba_codTara_codVariant`). Fiecare astfel de subclasă stochează toate datele specifice unei localizări particulare grupate sub formă de proprietăţi.

Exemple de astfel de subclase:

```
Eticheta
Eticheta_ro_RO
Eticheta_en_US
Eticheta_en_GB
```

Pentru a obţine o referinţă spre o astfel de subclasă, corespunzătoare unei anumite localizări, se poate folosi secvenţa care urmează:

```
Locale localizare =
    new Locale("ro", "RO");
ResourceBundle resursa =
    ResourceBundle.getBundle("Eticheta", localizare);
```

În interior, fiecare clasă `ResourceBundle` este structurată ca un vector de perechi cheie/valoare împreună cu metodele de acces spre acestea. O cheie identifică unic un element din colecţia păstrată într-un obiect `ResourceBundle` şi este întotdeauna un şir de caractere. O valoare este în general un şir, dar poate fi orice fel de obiect. Pentru o cheie, se poate obţine valoarea corespunzătoare printr-un apel de forma `getString("cheie")` sau `getObject("cheie")`. Pentru cea de a doua metodă trebuie să avem grijă să realizăm operaţia de conversie spre clasa care furnizează tipul obiectului.

Platforma Java oferă două clase `ListResourceBundle` şi `PropertyResourceBundle`, care extind clasa abstractă `ResourceBundle`, fiind implementări efective ale acesteia, uşor de folosit de programatori. Dacă cele două clase nu îndeplinesc cerinţele dezvoltatorilor (spre exemplu, timpi prea mari de execuţie), aceştia pot să-şi creeze propria clasă pentru gestionarea datelor sensibile la localizare prin extinderea clasei `ResourceBundle`. Pentru aceasta, trebuie suprascrise două metode ale acesteia, şi anume `handleGetObject(String cheie)`, care returnează valoarea corespunzătoare cheii `cheie`, şi `getKeys()`, care va returna o listă `Enumeration<String>`

a cheilor resursei. Prezentăm, în continuare, o astfel de implementare simplă care păstrează două chei, pentru mai multe chei fiind nevoie de o structură (cel mai bine s-ar preta un tablou `HashMap`):

```
// limba romana
public class ResursaProprie extends ResourceBundle {
  public Object handleGetObject(String cheie) {
    if (cheie.equals("da")) return "De acord";
    if (cheie.equals("nu")) return "Anuleaza";
    return null;
  }
  public Enumeration<String> getKeys() {
    Vector<String> v = new Vector<String>();
    v.add("da");
    v.add("nu");
    return v.elements();
  }
}

// limba engleza
public class ResursaProprieEn extends ResursaProprie {
  public Object handleGetObject(String cheie) {
    if (cheie.equals("da")) return "Ok";
    return null;
  }
  public Enumeration<String> getKeys() {
    Vector<String> v = new Vector<String>();
    v.add("da");
    v.add("nu");
    return v.elements();
  }
}
```

Atunci când se doreşte crearea unui buton, se va folosi un apel de forma:

```
JButton buton = new JButton(myResources.getString("da"));
```

Dacă am fi folosit forma clasică, nu am mai fi oferit suport aplicaţiei pentru internaţionalizare:

```
JButton buton = new JButton("De acord");
```

Dacă dorim să nu realizăm propriile noastre implementări ale clasei `ResourceBundle`, putem folosi o extensie a acesteia, şi anume clasa `PropertiesResourceBundle`, care are în spate un fişier text având numele clasei şi extensia `.properties` şi conţinând perechi cheie/valoare, unde şi cheia, şi valoarea sunt şiruri de caractere. Dacă se doreşte păstrarea de valori care sunt instanţe ale clasei `Object`, este indicată folosirea clasei `ListResource`. Fişierul text poate fi creat cu orice editor de texte, dar trebuie să aibă numele clasei `ResourceBundle` şi să aibă extensia `.properties`. Această clasă respectă principiile clasei `Properties`, care reprezintă un set de proprietăţi persistente (salvate pe suport). Întotdeauna trebuie creat un fişier implicit. Spre exemplu, pentru un buton prezentăm un exemplu de fişier implicit:

```
# fisierul ResourceBundle Buton.properties
da = De acord
nu = Anuleaza
```

Comentariile se scriu folosind simbolul # sau `;`. Proprietăţile se scriu una sub alta în forma `cheie=valoare`.

Pe lângă fişierul implicit, se vor crea noi fişiere cu proprietăţi pentru alte localizări, conţinând valorile traduse pentru aceleaşi chei. Sursa aplicaţiei nu se va modifica, deoarece accesul spre valoarea dependentă de localizare se va face doar prin intermediul cheii. Numele fişierului este format din numele fişierului implicit la care se adaugă localizarea şi extensia `.properties`. Spre exemplu, resursele pentru limba engleză vor fi memorate într-un fişier `Button_en.properties`:

```
# fisierul ResourceBundle Button_en.properties
da = OK
nu = CANCEL
```

Următoarea secvenţă descrie procesul de creare a unor butoane folosind fişiere cu proprietăţi:

```
Locale localizare = new Locale("ro","");
ResourceBundle text = ResourceBundle.getBundle("Button_ro",
localizare);
String mesajDa = text.getString(da);
JButton butonDa = new JButton(mesajDa);
String mesajNu = text.getString(nu);
JButton butonNu = new JButton(mesajNu);
```

Printr-un apel al metodei `getBundle(String numeResursa, Locale localizare)` se va căuta, mai întâi, dacă există definită o clasă `ResourceBundle` având numele `numeResursa`, iar dacă nu se găseşte, se va căuta un fişier de proprietăţi care are nume corespunzător. Pentru căutarea fişierelor se foloseşte un algoritm descris în *javadoc*, ultimul luat în calcul fiind fişierul implicit. Dacă nu se găseşte nici un fişier de proprietăţi, atunci se va arunca o excepţie `java.util.MissingResourceException`.

Pentru compatibilitate cu modul în care a fost implementat sistemul de proprietăţi (clasa `Properties`), fişierele cu proprietăţi corespunzătoare clasei `PropertyResourcesBundle` trebuie scrise folosind setul de caractere Latin-1, eventual cu secvenţe Unicode.

Vom prezenta, în continuare, clasa `ListResourceBundle`, reprezentând o altă metodă de a gestiona resursele, folosind de această dată fişiere `.class` salvate pe suport. Pentru fiecare localizare, se va crea câte o clasă extinzând `ListResourceBundle` şi având un nume corespunzător, care va fi mai apoi compilată, rezultând un fişier cu extensia `.class`. Clasele vor trebui să suprascrie metoda `getContents()` ce returnează un tablou ale cărui elemente sunt perechi cheie/valoare.

```
public class Butoane_ro extends ListResourceBundle {
  public Object[][] getContents() {
    return continut;
  }
  static final Object[][] continut =
    {{"da", "De acord"},
     {"nu", "Anuleaza"},
     {"cod", new Integer(40)}};
}

public class Butoane_fr extends ListResourceBundle {
  public Object[][] getContents() {
    return continut;
  }
  static final Object[][] continut =
    {{"nu", "Annuler"},
     {"stop", "Arreter"},
     {"cod", new Integer(20)}};
}
```

După compilare, vor rezulta două fişiere `Butoane_ro.class` şi `Butoane_fr.class`. Dacă sursele Java ale acestor clase sunt scrise cu un alt set de caractere, într-un editor de texte ce permite acest lucru, acest set trebuie specificat explicit în momentul compilării, pentru a se face corect conversia spre Unicode. Altfel, se consideră că sursele au fost scrise folosind setul de caractere al sistemului de operare (spre exemplu, pentru Windows, pentru localizarea *English*, setul *Latin-1*). Pentru a compila sursa scrisă utilizând diacritice româneşti, se va recurge la:

```
javac -encoding Cp1250 Butoane_ro.java
```

Un lucru demn de remarcat este faptul că pentru toate clasele, proprietăţile se moştenesc, astfel încât nu e nevoie să repetăm proprietăţile dacă sunt aceleaşi pentru mai multe localizări. Ierarhia este dată de numele pe care îl poartă grupurile de resurse, această ierarhie determinând algoritmul de căutare ascendentă a proprietăţilor.

15.6. Formatări

Diversele localizări implică şi folosirea diferitelor formatări pentru datele ce vor fi expuse utilizatorilor. Numerele, sistemul monetar, datele calendaristice, reprezentarea orei şi mesajele sunt cele mai importante categorii de date folosite în procesul de formatare. La baza formatării în limbajul Java stă clasa `Format` din pachetul `java.text`, din care sunt derivate toate celelalte clase folosite pentru aceasta.

15.6.1. Formatarea numerelor

Numerele sunt reprezentate diferit în lume. Spre exemplu, în Statele Unite, numărul 3,140 reprezintă trei mii o sută patruzeci, iar, în România, acest număr reprezintă trei şi o sută patruzeci. Echivalentul în România pentru acest număr este 3.140. Aplicaţiile internaţionalizate vor oferi utilizatorilor formatarea corectă a numerelor în funcţie de localizarea curentă. Diferenţele constau în simbolul zecimal, separatorii pentru grupuri, numărul de cifre după simbolul zecimal şi modul de folosire a cifrei zero înainte de simbolul zecimal.

Clasa folosită pentru formatarea numerelor este `java.text.NumberFormat`. Pentru a obţine o instanţă a acestei clase corespunzătoare unei localizări se va folosi un apel `NumberFormat.getNumberInstance(Locale localizare)`. Având instanţa `NumberFormat`, formatarea se poate realiza printr-un apel al metodelor `format(long număr)` sau `format(double număr)` pentru a obţine un şir de caractere conţinând numărul formatat, şir care va fi prezentat utilizatorilor.

Secvenţa care urmează, aplicată pentru mai multe localizări, va determina afişarea diferitelor formatări pentru un acelaşi număr:

```
Locale localizare; // o localizare oarecare
Double numar = new Double(67947.123);
NumberFormat formatare =
  NumberFormat.getNumberInstance(localizare);
String  rezultat = formatare.format(numar);
System.out.println(rezultat + " " + localizare.toString());
```

Rezultatele obţinute vor fi de genul:

```
67 947,123 fr_FR
67.947,123 de_DE
67,947.123 en_US
67.947,123 ro_RO
```

15.6.2. Reprezentarea monetară

Pentru fiecare localizare, sumele de bani se vor reprezenta într-un anumit mod. De asemenea, simbolurile monetare sunt diferite de la o ţară la alta. Formatarea sumelor de bani se realizează în acelaşi mod ca şi formatarea numerelor, cu singura deosebire că se va folosi un apel `getCurrencyInstance()`. De remarcat faptul că se face doar afişarea unei aceleaşi sume în diferite localizări, fără a face conversiile valutare corespunzătoare.

```
Locale localizare; // o localizare oarecare
Double numar = new Double(67947.123);
NumberFormat formatare =
  NumberFormat.getCurrencyInstance(localizare);
String  rezultat = formatare.format(numar);
System.out.println(rezultat + " \t" + localizare.toString());
```

Similar obţinem:

```
67 947,123 F      fr_FR
67.947,123 DM     de_DE
$67,947.123       en_US
67.947,123 LEI    ro_RO
```

15.6.3. Formatarea datei calendaristice şi a orei

Datele calendaristice sunt păstrate intern, independent de orice localizare, sub formă de obiecte `Date`, dar afişarea acestora se poate face doar ca şiruri de caractere conform unei anumite localizări. Pentru a obţine anul, luna, săptămâna dintr-un obiect `Date` se va folosi un obiect `Calendar`. Formatarea datei încapsulate într-un obiect `Date` se realizează cu ajutorul clasei `java.text.DateFormat` într-o manieră asemănătoare cu formatarea numerelor.

```
Locale localizare; // o localizare oarecare
Date azi = new Date();
DateFormat formatare =
   DateFormat.getDateInstance(DateFormat.DEFAULT, localizare);
String data = formatare.format(azi);
System.out.println(data + " \t" + localizare);
```

Efectul acestei secţiuni de cod în cazul mai multor localizări îl puteţi vedea în continuare:

```
07.01.2003      ro_RO
7 janv. 2003    fr_FR
Jan 7, 2003     en_US
```

Afişarea datei se poate face cu mai multe stiluri. Clasa `DateFormat` suportă mai multe stiluri, şi anume `DEFAULT`, `SHORT`, `MEDIUM`, `LONG`, `FULL` introduse ca parametru de intrare pentru metoda `getDateInstance()`.

Pentru formatarea orei se va folosi metoda `getTimeInstance()`, ca în exemplul care urmează:

```
DateFormat formatare =
   DateFormat.getTimeInstance(DateFormat.DEFAULT, localizare);
```

Clasele `java.text.SimpleDateFormat` şi `java.text.DateFormatSymbols` pot fi utilizate pentru a crea formatări proprii pentru date calendaristice, formatări care să fie folosite, spre exemplu, în localizările specificate de programator.

15.6.4. Formatarea mesajelor

Această secţiune se ocupă de formatarea mesajelor compuse. Prin mesaj compus vom înţelege un mesaj care conţine atât părţi fixe, cât şi părţi variabile. În textul de mai jos vom marca părţile variabile ale unui mesaj prin subliniere:

Peste 300.000 de persoane au vizitat situl Web www.infoiasi.ro până la data de 1 ianuarie 2007.

Părţile variabile ale unui mesaj compus pot fi date, şiruri, numere, unităţi monetare şi procente. Pentru a formata un mesaj compus independent de localizare, se va construi un şablon ce va fi matriţa cu care se creează un obiect `MessageFormat`. Şablonul are un anumit mod de construcţie, conform descrierii pe care o găsiţi în documentaţia clasei `MessageFormat`.

Prezentăm un exemplu de şablon pentru mesajul compus prezentat anterior, pe care presupunem că îl păstrăm în clasa `Mesaje`, extensie a clasei `ResourceBundle`:

```
sablon = Peste {2, number, integer} persoane au vizitat
         situl Web {0} până la data de {1, date, long}
```

O instanţă a clasei `Mesaje` se obţine astfel:

```
ResourceBundle mesaje =
  ResourceBundle.getBundle("Mesaje",
          new Locale("ro","RO"));
```

Se observă că acest mesaj este o reprezentare a şirului iniţial, în care s-au înlocuit secvenţele variabile cu acolade, având drept conţinut numărul secvenţei variabile urmat de tipul ei, respectiv modul de formatare. Se poate observa că nu este o anumită ordine a numerotării secvenţelor variabile.

Crearea unui obiect `MessageFormat` are forma:

```
MessageFormat formatare = new MessageFormat("");
formatare.setLocale(new Locale("ro","RO"));
```

Pentru a configura obiectul `formatare` astfel încât să se poată formata folosind şablonul depozitat în clasa `Mesaje`, acesta din urmă se va extrage din `ResourceBundle` printr-un apel `getString()` şi, mai apoi, va fi folosit printr-un apel `applyPattern()`, aşa cum se poate vedea în secvenţa care urmează:

```
formatare.applyPattern(mesaje.getString("sablon"));
```

Introducerea datelor variabile în şir se va realiza prin încapsularea lor într-un vector de obiecte, ca în exemplul următor:

```
Object[] argumenteVariabile =
  {"www.infoiasi.ro", new Date(), 300000};
```

Ordinea în care se introduc elementele în tablou este importantă şi trebuie să fie în concordanţă cu şablonul depozitat în `ResourceBundle`. Pentru a popula un şablon cu date variabile, se vor folosi apeluri de forma:

```
String rezultat = formatare.format(argumenteVariabile);
```

Va rezulta exact şirul iniţial. Foarte interesant este faptul că, stocând un şablon într-o clasă `ResourceBundle`, îi putem aplica uşor aceleaşi date iniţiale. Putem, spre exemplu, să afişăm facil un mesaj în altă limbă decât cea implicită, cu formatarea corespunzătoare. Spre exemplu, folosind șablonul `Over {2, number, integer} people have visited {0} website since {1, date, long}` obținem șirul `Over 300000 people have visited www.infoiasi.ro website since January 1, 2007`, folosind secvența de cod:

```
Locale localizare1 = new Locale("en","US");
ResourceBundle mesaje1 =
  ResourceBundle.getBundle("Mesaje", localizare1);
formatare1.setLocale(localizare1);
formatare1.applyPattern(mesaje1.getString("sablon"));
String rezultat1 = formatare1.format(argumenteVariabile);
System.out.println(rezultat1);
```

15.7. Lucrul cu fonturi

În ceea ce privește fonturile, platforma Java dă posibilitatea utilizării a două tipuri de fonturi, și anume fonturi fizice și fonturi logice. Fonturile fizice pot fi *True Type* sau *PostScript Type 1*. JRE caută fonturile fizice fie în directorul cu fonturi al sistemului de operare (spre exemplu, `Windows/Fonts` pentru sistemul de operare Windows), fie în directorul `lib/fonts` din directorul în care este instalat JRE. În unul dintre aceste două directoare utilizatorii pot să-și instaleze propriile fonturi.

Implicit, platforma Java oferă o serie de fonturi fizice din familia *Lucida Sans*. Acestea vin împreună cu JRE și nu sunt dependente de sistemul de operare. Folosirea lor oferă posibilitatea unui *look-and-feel* identic pentru toate sistemele de operare peste care este instalată platforma Java. Acestea sunt *Lucida Sans*, *Lucida Sans Typewriter* și *Lucida Bright* și permit folosirea stilurilor *regular*, *bold*, *italic*, *italic-bold*, fiind instalate în directorul `lib/fonts`.

Fonturile logice reprezintă asocieri spre fonturile fizice deja existente. În general, un font logic se asociază la mai multe fonturi fizice, oferind astfel un set mai mare de caractere la un moment dat. Asocierile sunt păstrate în fișiere având numele `font.properties` stocate în directorul `lib` al JRE. Din această categorie de fonturi, orice implementare a platformei Java trebuie să ofere suport pentru *Serif*, *SansSerif*, *Monospaced*, *Dialog* și *DialogInput*. Utilizatorul poate interveni asupra unor fișiere `font.properties` deja existente sau poate să creeze altele, după propriile preferințe.

Utilizatorul poate afla, la un moment dat, toate fonturile disponibile pentru mediul de lucru curent și poate utiliza fonturile folosind numele lor, ca, de exemplu, *Times Roman* sau *Tahoma*. De asemenea, se pot folosi fonturi neinstalate încă, folosind apeluri ale metodei `create()` a clasei `Font`. Pentru mai multe informații puteți consulta secțiunea destinată desenării din capitolul „Swing”.

15.8. Introducerea textului

Keyboard Layout şi *Input Method* sunt cele două modalităţi de a introduce text drept conţinut pentru o componentă text într-un mod mai compex decât simpla afişare a caracterului corespunzător tastei apăsate. Acest lucru este valabil pentru cazul în care un caracter nu poate fi reprezentat printr-o singură apăsare de tastă. Astfel, se pot introduce toate caracterele unei limbi, care pot fi în număr considerabil, folosind tastatura ce are un număr finit de taste.

În cazul *Keyboard Layout* se realizează o reasociere a tastelor pentru a se putea introduce alte caractere, diferite de cele desenate pe taste. În Java, sistemul pentru *Keyboard Layout* este total dependent de sistemul de operare peste care este instalată platforma. Spre exemplu, în Windows, schimbarea *Keyboard Layout* se va realiza din *Control Panel/Keyboard/Language* sau din *Language Bar*, dacă aceasta este disponibilă (eventual folosind *shortcut*-uri definite în prealabil). Dacă de aici se va alege tastatura românească, atunci orice componentă Swing va accepta introducerea diacriticelor prin apăsarea tastelor corespunzătoare. Trebuie reţinut faptul că toate componentele Swing codează textul folosind Unicode, astfel încât programatorul este responsabil, mai apoi, pentru modul în care transferă textul înspre şi dinspre componentele Swing, pentru a nu apărea probleme de codare.

În ceea ce priveşte *Input Method*, se poate spune că există mai multe metode de a introduce text de la tastatură. Prin metoda directă (engl. *on the spot*, *inline*) se permite introducerea unei secvenţe de caractere direct în componentele text, dar marcate prin subliniere, astfel că scrierea lor efectivă va avea loc numai după o confirmare. Acesta este modul implicit de introducere pentru componentele care au asociată o metodă de introducere. Prin metoda indirectă (engl. *below the spot*), se va activa o fereastră auxiliară pentru compunerea textului (engl. *composition*), în imediata apropiere a poziţiei curente a cursorului (neieşirea ei din ecran este asigurată). Prin apăsarea unei combinaţii de taste prestabilite, caracterele introduse în fereastra de compoziţie sunt convertite într-o altă secvenţă sau un singur caracter care va fi transferat în componenta text.

Java oferă posibilitatea atât a folosirii metodelor de introducere oferite de sistemul de operare, cât şi a definirii de către dezvoltatori a propriilor metode de introducere a textului. Utilizatorilor le este pus la dispoziţie un întreg cadru (engl. *framework*) de lucru în acest ultim scop, care poartă denumirea de *Input Method Framework*. Acest cadru cuprinde un API-client conţinut în pachetul `java.awt.im` prin care se poate realiza programatic comunicarea dintre componente şi ferestrele de compoziţie corespunzătoare metodei curente de introducere a datelor. De asemenea, mai cuprinde un motor SPI cuprins în pachetul `java.awt.im.spi` care oferă posibilitatea creării de metode de introducere independente de o anumită aplicaţie Java, ce vor fi, mai apoi, folosite în diferite medii de rulare prin simpla ataşare a fişierului JAR care le conţine în directorul `lib/ext`. La baza acestui cadru stă o specificaţie pe care o puteţi găsi în ghidul ataşat documentaţiei standard (`docs/guide/imf`) a JRE.

Metodele de introducere pot fi folosite nu numai pentru introducerea de caractere pentru diverse limbi, ci şi în alte scopuri. A se vedea modalitatea prin care utilizatorul este asistat în procesul de scriere a codului-sursă într-un mediu de programare vizual. În momentul editării, atunci când numele metodei a fost scris, prin apăsarea unei combinaţii de taste va apărea o listă derulantă a tuturor supraîncărcărilor cunoscute ale metodei, dintre care utilizatorul poate alege una.

Exemple de metode de introducere puteţi găsi în directorul documentaţiei: `docs/guide/imf/spi_sample/index.html`.

15.9. Paşii necesari pentru internaţionalizarea aplicaţiilor

Pentru orice aplicaţie există anumite caracteristici care au probabilitatea cea mai mare de a fi subiectul internaţionalizării, şi anume:

- mesajele
- etichetele componentelor grafice
- ajutorul online
- sunetele
- culorile
- graficele
- pictogramele
- datele calendaristice
- timpul
- numerele
- unităţile monetare
- unităţile de măsură
- numerele de telefon
- titlurile personale şi onorifice
- adresa poştală
- poziţionarea în pagină

Aceste informaţii sunt izolate, în general, de aplicaţie (nu sunt codate în aceasta), tocmai pentru a o putea internaţionaliza.

În momentul internaţionalizării, apar şi anumite probleme. Spre exemplu, deoarece setul de caractere poate cuprinde şi diacritice, nu se mai pot face comparaţii pentru stabilirea literelor de genul:

```
if ((caracter >= 'a' && caracter <= 'z')    { . . . }
```

Determinarea literelor, cifrelor, semnului pentru spaţiu se realizează folosind standardul Unicode prin metode statice de genul `isCaracter(ch)`, `isDigit(ch)`, `isSpaceBar(ch)` oferite de clasa `Character`.

De asemenea, compararea şirurilor nu se mai poate realiza prin metodele clasei `String`: `equals()` şi `compareTo()` atunci când dorim să realizăm sortări, din cauza caracterelor noi care apar în alfabetele diferite de cel englez. În aplicaţiile internaţionalizate, comparaţiile se realizează prin intermediul clasei `Collator`, mai precis prin intermediul metodei `compare()`. De exemplu, pentru compararea şirurilor conţinând caractere româneşti:

```
Locale diacritice = new Locale("ro", "RO");
Collator pentruRomana = Collator.getInstance(diacritice);
int rezultat = pentruRomana.compare("păr", "par");
```

15.10. Concluzii

Pentru început, au fost descrise aspecte generale legate de internaţionalizarea (*i18n*) şi localizarea aplicaţiilor (*l10n*). S-a discutat apoi despre seturi de caractere, realizarea de conversii între acestea, despre codificări şi, în special, despre Unicode, care reprezintă setul de caractere folosit intern de maşina virtuală Java.

S-au pus problemele localizării şi a modului în care acest principiu este codat în aplicaţiile Java prin intermediul clasei `Locale`. Ulterior, s-a prezentat modalitatea de a despărţi datele de codul aplicaţiei, pentru a putea astfel adapta aplicaţia la mai multe localizări, prin folosirea clasei `ResourceBundle` şi, mai ales, a derivatelor din aceasta pe care platforma Java le pune la dispoziţia dezvoltatorilor.

Formatarea datelor conform localizărilor, şi anume formatarea numerelor, unităţilor monetare, datelor calendaristice şi a orei, respectiv formatarea mesajelor reprezintă un aspect foarte important al procesului de internaţionalizare.

Am descris câteva aspecte legate de fonturi şi metodele de introducere a textului, fără a intra în amănunte, rămânându-le cititorilor să studieze API-urile corespunzătoare.

15.11. Test-grilă

Întrebarea 15.11.1. Ce reprezintă internaţionalizarea aplicaţiilor?

a) Scrierea de aplicaţii care să permită folosirea meniurilor scrise în limba română.
b) Un proces de proiectare a aplicaţiilor, astfel încât acestea să poate accesa fişiere scrise cu diverse seturi de caractere.
c) Un proces de proiectare a aplicaţiilor, astfel încât acestea să se poată adapta la diverse localizări.

Întrebarea 15.11.2. Ce reprezintă Unicode?

a) Un set de caractere.
b) Un criptosistem.
c) O aplicaţie folosită pentru internaţionalizarea aplicaţiilor.
d) Un cod binar.

Întrebarea 15.11.3. Care este funcţionalitatea utilitarului `native2ascii`?

a) Converteşte fişiere scrise folosind codarea UTF-16 în fişiere folosind codarea UTF-8.
b) Conversie spre setul Latin-1 din orice alt set de caractere.
c) Localizează fişierele având numele drept parametru.
d) Traduce în Unicode sursele încărcate în maşina virtuală.

Întrebarea 15.11.4. Fie secvenţa de cod:

```
Locale.setDefauld(new Locale("ro", "RO"));
Locale loc=new Locale("en", "US");
System.out.println("Localizare: "+loc.toString());
```

Ce se va afişa în urma execuţiei codului?

a) `Localizare: ro_RO.`
b) `Localizare: româna:Romania.`
c) `Localizare: en_US.`
d) Se va arunca o excepţie.

Întrebarea 15.11.5. Considerăm un fişier cu numele `Siruri.properties` conţinând următoarea informaţie:

```
da = DeAcord
nu = Anuleaza
```

şi secvenţa de cod:

```
Locale localizare= new Locale("ro","Ro");
ResourceBundle text =
   ResourceBundle.getBundle("Buton", localizare);
Sytem.out.println("Mesaj: " + text.getString(da)
   + "/" + text.getString(nu));
```

Ce se va întâmpla după execuţia codului?

a) Se va genera o excepţie `MissingResourceException.`
b) Se va afişa `Mesaj: DeAcord/Anuleaza.`
c) Se va afişa `Mesaj: OK/Cancel.`
d) Se va afişa `Mesaj: da/nu.`

Întrebarea 15.11.6. Fie secvenţa de cod:

```
Locale localizare = Locale.getDefault();
Float numar = 845.12f;
NumberFormat f =
   NumberFormat.getCurrencyInstance(localizare);
System.out.println(f.format(numar));
```

În urma execuţiei codului, care dintre efectele prezentate în continuare este valabil întotdeauna?

a) Se va afişa 845,12.
b) Se va afişa: 845,12 LEI.
c) Se va arunca o excepţie.
d) Se va afişa suma 845.12, în localizarea implicită.
e) Se va afişa numărul real 845.12, în localizarea implicită.

Întrebarea 15.11.7. Ce reprezintă metodele de introducere a textului?

a) Metode prin care putem introduce diacritice.
b) Metode prin care putem asocia alte caractere butoanelor tastaturii.

c) Metode de a introduce date în aplicaţiile Java.
d) Metode prin care putem formata textul din componentele Swing.

15.12. Exerciţii propuse spre implementare

Exerciţiul 15.12.1. Implementaţi o aplicaţie Java care să realizeze conversii între diversele codări (UTF-8, UTF-16, UTF-32) ale setului de caractere Unicode.

Exerciţiul 15.12.2. Să se scrie o clasă Java care suprascrie `ResourceBundle`, permiţând salvarea grupurilor de date în fişiere XML, de unde vor putea fi folosite ulterior.

Exerciţiul 15.12.3. Dezvoltaţi o aplicaţie care face conversii ale fişierelor scrise cu orice tip de caractere spre fişiere scrise folosind Unicode.

15.13. Proiecte propuse spre implementare

Proiectul 15.13.1. (*Dicţionar*) Realizaţi o aplicaţie având funcţionalitatea unui dicţionar român-englez, englez-român. Acesta va ţine datele separat de cod, sub formă de fişiere XML sau în baze de date. Va oferi posibilitatea introducerii de noi cuvinte, modificarea vechilor cuvinte etc.

Proiectul 15.13.2. (*Translator*) Creaţi o aplicaţie care să traducă expresii standard în mai multe limbi. Se vor folosi formatarea mesajelor şi seturi de caractere specifice fiecărei limbi în parte.

Proiectul 15.13.3. Realizaţi o aplicaţie care să reprezinte un *plug-in* pentru orice altă aplicaţie, permiţând introducerea de diacritice româneşti folosind secvenţe Unicode. Pentru documentare, se poate consulta *Input Method Framework*.

Proiectul 15.13.4. Realizaţi o aplicaţie care să reprezinte un gestionar de resurse. Va permite gruparea acestora şi crearea lor, iar mai apoi includerea lor în aplicaţii într-un mod uniform.

Proiectul 15.13.5. Realizaţi o aplicaţie care să permită internaţionalizarea surselor Java care nu au fost proiectate de la început conform principiilor internaţionalizării. Aplicaţia va determina şirurile de caractere din surse şi va oferi posibilitatea salvării acestora în fişiere, împreună cu traduceri ale lor pentru diverse localizări, de unde vor fi mai apoi încărcate folosind `ResourceBundle`, în funcţie de localizarea folosită curent de aplicaţie.

Proiectul 15.13.6. Construiţi o aplicaţie care să poarte conversaţii inteligibile cu cei ce o accesează. Aplicaţia va detecta singură limba în care i se adresează clientul şi va fi destul de instruită pentru a învăţa cuvinte şi expresii introduse de acesta.

Bibliografie

Referinţe cărţi:

[Ben90] – BenAri, M: *Principles of Concurrent and Distributed Programming*, Prentice Hall (1990).

[BeP99] – Bell, D., Perr M.: *Java for Students*, ediţia a doua, Prentice Hall (1999).

[Bud00] – Budd, T.: *Understanding ObjectOriented Programming with JAVA*, Addison-Wesley (2000).

[Bur01] – Buraga, S.: *Tehnologii Web*, Editura Matrix Rom (2001).

[Bur02] – Buraga, S.: *Proiectarea siturilor Web*, Editura Polirom (2002).

[Bur06] – Buraga, S.: *Tehnologii XML*, Editura Polirom (2006).

[BTOT] – Buraga, S., Tarhon-Onu, V., Tanasă, Ş.: *Programare Web în bash şi Perl*, Polirom (2002).

[CGI00] – Chan, M.C., Griffith, S.W., Iasi, A.F.: *JAVA. 1001 secrete pentru programatori*, Editura Teora (2000).

[Eck98] – Eckel, B.: *Thinking in JAVA*, Prentice Hall (1998).

[GJS96] – Gosling, J., Joy, B., Stelle, G.: *The Java Language Specification*, USA (1996).

[GrK97] – Grand, M., Knudsen, J.: *JAVA. Fundamental Classes Reference*, O'Reilly (1997).

[Hor00] – Horton, I.: *Beginning Java 2. JDK 1.3*, ediţia a treia, Wrox Press Ltd (2000).

[HSH99] – Hughes, M., Shoffner, M., Hamner, D.: *JAVA. Network Programming*, Manning (1999).

[LaO99] – Lambert, K.A., Osborne, M.: *Java. A Framework for Programming and Problem Solving*, PWS Publishing (1999).

[NoS96] – Norton, P., Stanek, W.: *Peter Norton's Guide to Java Programming*, Samsa Press (1996).

[RHE99] – Roberts, S., Heller, P., Ernest, M.: *JAVA 2 Certification Study Guide*, Sybex (1999).

[Tan05] – Tanasă, Ş., Olaru, C.: *Dezvoltarea aplicaţiilor Web folosind Java*, Editura Polirom (2005).

[Vad99] – Văduva, C.M.: *Programarea în JAVA*, Microinformatica (1999).

[Van96] – Vandenburg, G.L. *et al.*: *Tricks of the Java Programming Gurus*, Samsa Publishing (1996).

[WiN96] – Winston, P.H., Narasimhan, S.: *On to JAVA*, Addison-Wesley (1996).

Referinţe Web:

Eckel, B., *Thinking in Java*, ediţia a treia: *http://www.mindview.net/Books/*

Robinson, M., Vorobiev, P., *Swing Release Java 2*: *http://developer.java.sun.com/developer/Books/swing2/*

* * *, *JDK 6 Documentation*: *http://java.sun.com/javase/6/docs/*

* * *, *Servlet Documentation: http://java.sun.com/products/servlet/docs.html*

* * *, *JSP Documentation: http://java.sun.com/products/jsp/docs.html*

* * *, *The Source for Java Developers: http://java.sun.com/*

* * *, *Java World: http://www.javaworld.com/*

* * *, *Javalobby: http://www.javalobby.org/*

* * *, *Application Development Trends: http://www.javareport.com/*

* * *, *W3C Technical Reports and Publications: http://www.w3c.org/TR/*

* * *, *Extensible Markup Language (XML): http://www.w3c.org/XML/*

* * *, *The Extensible Stylesheet Language (XSL): http://www.w3c.org/Style/XSL/*

* * *, *JDBC Tutorial: http://java.sun.com/docs/books/tutorial/jdbc/index.html*

* * *, *JDBC API Documentation: http://java.sun.com/j2se/1.4.1/docs/guide/jdbc/index.html*

* * *, *Swing tutorial: http://java.sun.com/docs/books/tutorial/uiswing/index.html*

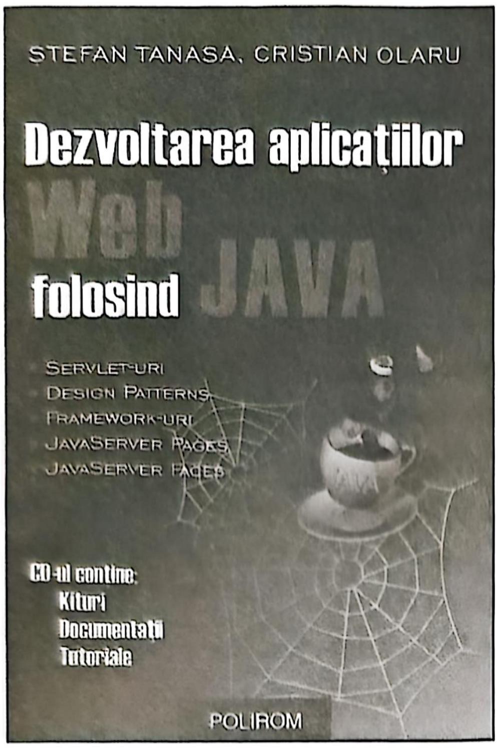
* * *, *Tutorial de grafică 2D: http://java.sun.com/docs/books/tutorial/2d/index.html*

* * *, *JHELP User Guide: http://java.sun.com/products/javahelp/*

* * *, *Internationalization Tutorial: http://java.sun.com/docs/books/tutorial/i18n/index.html*

* * *, *Unicode: http://www.unicode.org/*

* * *, *Tomcat: http://tomcat.apache.org/*


ȘTEFAN TANASA, CRISTIAN OLARU
Dezvoltarea aplicațiilor
Web
folosind JAVA
SERVLET-URI
DESIGN PATTERNS
FRAMEWORK-URI
JAVASERVER PAGES
JAVASERVER FACES
CD-ul conține:
Kituri
Documentații
Tutoriale
POLIROM

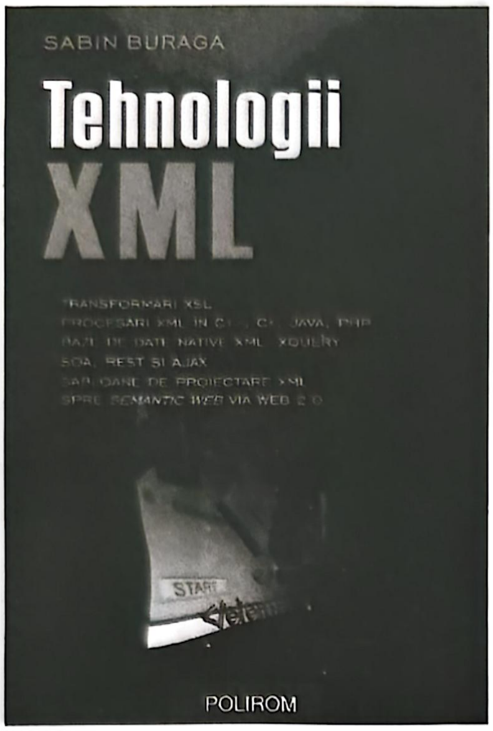
Ştefan Tanasă, Cristian Olaru

Dezvoltarea aplicaţiilor Web folosind JAVA

(Cartea include CD)

Limbajul Java oferă un suport excelent pentru realizarea de aplicaţii Web complexe şi este preferat de programatori pentru uşurinţa dezvoltării, întreţinerii şi actualizării acestora. Printre atuurile sale se numără independenţa faţă de platformă, posibilităţile de reutilizare cu uşurinţă a codului, numeroase instrumente gratuite pentru dezvoltare şi o bogată documentaţie disponibilă pe internet. Adresată programatorului cu minime cunostinţe de Java şi tehnologii Web, cartea propune spre rezolvare problema creării unui magazin de comerţ electronic online. Sunt prezentate elemente frecvent întâlnite în aplicaţiile Web: autentificarea utilizatorilor, conectarea la serverul de baze de date, gestiunea informaţiilor şi administrarea de la distanţă, luând în calcul toate etapele, de la proiectare şi implementare până la gestionare, îmbunătăţire şi extindere.

Servlet-uri • JavaServer Pages • *Framework*-uri Web • JavaServer Faces



Sabin Buraga

Tehnologii XML

Cele mai importante tendinţe în utilizarea, proiectarea, procesarea şi conceptualizarea datelor prin suita de tehnologii XML (*eXtensible Markup Language*) sunt obiectul acestui volum, care îl va conduce pe cititor spre viitorul spaţiului World Wide Web. Subiectele atinse includ diverse aspecte ale familiei XML şi soluţiile pentru modelarea şi stocarea conţinutului semistructurat: baze de date native XML, şabloane de proiectare XML, interacţiunea Web prin AJAX. Sunt descrise posibilităţile de procesare XML prin diferite metode, de redare a conţinutului prin transformări XSL (*Extensible Stylesheet Language*), de validare a documentelor şi de creare a serviciilor Web, indicându-i-se cititorului cele mai bune practici de utilizare. De asemenea, cartea punctează conceptele de bază şi cele mai populare tehnologii ale Web-ului, subliniind direcţiile de evoluţie către *Data Web*, *Social Web* şi ceea ce va fi *Semantic Web*.

Arhitectura spaţiului WWW • Protocolul HTTP • *Cookie*-uri şi sesiuni • Date semistructurate • Familia XML: sintaxă, proiecte, transformări, procesare, validare • Servere de baze de date native XML • Interogări XQuery • Servicii Web • Transferul asincron prin AJAX • Aplicaţii Web fără fir • De la Web la Data Web şi Semantic Web • Specificarea meta-datelor şi ontologiilor Web • Inginerie Web

www.polirom.ro

Bun de tipar: noiembrie 2011. Apărut: 2011
Editura Polirom, B-dul Carol I nr. 4 • P.O. Box 266
700506, Iaşi, Tel. & Fax: (0232) 21.41.00 ; (0232) 21.41.11 ;
(0232) 21.74.40 (difuzare); E-mail: office@polirom.ro
Bucureşti, Splaiul Unirii nr. 6, bl. B3A, sc. 1, et. 1,
sector 4, 040031, O.P. 53 • C.P. 15-728
Tel.: (021) 313.89.78; E-mail: office.bucuresti@polirom.ro

Tiparul executat la S.C. LUMINA TIPO s.r.l.
str. Luigi Galvani nr. 20 bis, sect. 2, Bucureşti
Tel./Fax: 211.32.60, 212.29.27, E-mail: office@luminatipo.com

ŞTEFAN TANASĂ, ŞTEFAN ANDREI, CRISTIAN OLARU

JAVA

de la 0 la expert

Java este un limbaj de programare orientat obiect deja consacrat. Datorită dinamismului său, este utilizat în scrierea celor mai multe aplicaţii distribuite, noile evoluţii tehnologice validând folosirea sa pentru software-ul dispozitivelor portabile (telefoane mobile, agende electronice, palmtop-uri).

Cititorului i se oferă un ghid complet de învăţare, volumul incluzând cunoştinţe de bază, dar şi de nivel mediu şi avansat referitoare la limbajul Java. Lucrarea este bogată în exemple de programe comentate, fiecare capitol terminându-se cu realizarea unei aplicaţii concrete, pentru a ilustra mai bine noţiunile şi instrumentele de programare puse la dispoziţie de acest limbaj. Capitolele au o structură similară, începând cu secţiunea de cuvinte-cheie şi încheindu-se cu un scurt rezumat, un test-grilă şi exerciţii şi proiecte propuse spre implementare. Volumul se adresează studenţilor şi profesorilor de la facultăţile de informatică şi tuturor celor pasionaţi de dezvoltarea de aplicaţii Web de anvergură.

- FUNDAMENTELE LIMBAJULUI JAVA
- FIRE DE EXECUŢIE
- PROGRAMAREA REŢELELOR
- JAVA SERVER PAGES (JSP)
- PROCESAREA DOCUMENTELOR XML
- INTERACŢIUNEA CU UTILIZATORUL PRIN SWING

ISBN 978-973-46-2405-8

EDITURA POLIROM
www.polirom.ro